

Event-Oriented Dynamic Adaptation of Workflows: Model, Architecture, and Implementation

Von der Fakultät für Mathematik und Informatik
der Universität Leipzig
angenommene

D I S S E R T A T I O N

zur Erlangung des akademischen Grades

Doktor-Ingenieur
(Dr.-Ing.)

im Fachgebiet
Informatik

vorgelegt

von Diplom-Mathematiker R o b e r t M ü l l e r

geboren am 26.4.1966 in Witten/Ruhr

Die Annahme der Dissertation haben empfohlen:

1. Prof. Dr. Erhard Rahm, Fakultät für Mathematik und Informatik, Universität Leipzig
2. Prof. Dr. Gerhard Brewka, Fakultät für Mathematik und Informatik, Universität Leipzig
3. Prof. Dr. Peter Dadam, Fakultät für Informatik, Universität Ulm

Die Verleihung des akademischen Grades erfolgt auf Beschluss des Rates der Fakultät für Mathematik und Informatik vom 16.12.2002 mit dem Gesamtprädikat „magna cum laude“

Abstract

Workflow management is widely accepted as a core technology to support long-term business processes in heterogeneous and distributed environments. However, conventional workflow management systems do not provide sufficient flexibility support to cope with the broad range of failure situations that may occur during workflow execution. In particular, most systems do not allow to dynamically adapt a workflow due to a failure situation, e.g., to dynamically drop or insert execution steps.

As a contribution to overcome these limitations, this dissertation introduces the agent-based workflow management system AGENTWORK. AGENTWORK supports the definition, the execution and, as its main contribution, the event-oriented and semi-automated dynamic adaptation of workflows. Two strategies for automatic workflow adaptation are provided. *Predictive adaptation* adapts workflow parts affected by a failure in advance (predictively), typically as soon as the failure is detected. This is advantageous in many situations and gives enough time to meet organizational constraints for adapted workflow parts. *Reactive adaptation* is typically performed when predictive adaptation is not possible. In this case, adaptation is performed when the affected workflow part is to be executed, e.g., before an activity is executed it is checked whether it is subject to a workflow adaptation such as dropping, postponement or replacement. In particular, the following contributions are provided by AGENTWORK:

A Formal Model for Workflow Definition, Execution, and Estimation: In this context, AGENTWORK first provides an object-oriented workflow definition language. This language allows for the definition of a workflow's control and data flow. Furthermore, a workflow's cooperation with other workflows or workflow systems can be specified. Second, AGENTWORK provides a precise workflow execution model. This is necessary, as a running workflow usually is a complex collection of concurrent activities and data flow processes, and as failure situations and dynamic adaptations affect *running* workflows. Furthermore, mechanisms for the estimation of a workflow's future execution behavior are provided. These mechanisms are of particular importance for predictive adaptation.

Mechanisms for Determining and Processing Failure Events and Failure Actions: AGENTWORK provides mechanisms to decide whether an event constitutes a failure situation and what has to be done to cope with this failure. This is formally achieved by evaluating event-condition-action rules where the event-condition part describes under which condition an event has to be viewed as a failure event. The action part represents the necessary actions needed to cope with the failure. To support the *temporal* dimension of events and actions, this dissertation provides a novel event-condition-action model based on a temporal object-oriented logic.

Mechanisms for the Adaptation of Affected Workflows: In case of failure situations it has to be decided how an affected workflow has to be dynamically adapted on the node and edge level. AGENTWORK provides a novel approach that combines the two principal strategies reactive adaptation and predictive adaptation. Depending on the context of the failure, the appropriate strategy is selected. Furthermore, *control flow* adaptation operators are provided which translate failure actions into structural control flow adaptations. *Data flow* operators adapt the data flow after a control flow adaptation, if necessary.

Mechanisms for the Handling of Inter-Workflow Implications of Failure Situations: AGENTWORK provides novel mechanisms to decide whether a failure situation occurring to a workflow affects other workflows that communicate and cooperate with this workflow. In particular, AGENTWORK derives the *temporal* implications of a dynamic adaptation by estimating the duration that will be needed to process the changed workflow definition (in comparison with the original definition). Furthermore, *qualitative* implications of the dynamic change are determined. For this purpose, so-called *quality measuring objects* are introduced.

All mechanisms provided by AGENTWORK include that users may interact during the failure handling process. In particular, the user has the possibility to reject or modify suggested workflow adaptations.

A Prototypical Implementation: Finally, a prototypical CORBA-based implementation of AGENTWORK is described. This implementation supports the integration of AGENTWORK into the distributed and heterogeneous environments of real-world organizations such as hospitals or insurance business enterprises.

Table of Content

<i>Table of Content</i>	<i>iii</i>
<i>Foreword and Acknowledgements</i>	<i>ix</i>
CHAPTER 1 <i>Introduction and Problem Description</i>	<i>1</i>
1.1. Workflow Management	2
1.2. Failure Handling in Workflow Management Systems	5
1.2.1. Device, System, and Transaction Failures	6
1.2.2. Logical Failures	7
1.2.3. Control Flow Failures	8
1.3. Requirements of Handling Control Flow Failures for Workflow Systems	10
1.4. Contributions of the Thesis	16
1.5. The HematoWork System: Workflow Management in Hemato-Oncology	18
1.6. Structure of the Thesis	20
CHAPTER 2 <i>Related Work</i>	<i>21</i>
2.1. Commercial Workflow Management Systems	22
2.2. Advanced Transaction Models	24
2.2.1. Partial Backward Recovery and Compensation-Based Recovery	25
2.2.2. Forward Recovery	26
2.2.3. Discussion	27
2.3. Exception Handling in Programming Languages	28
2.4. Workflow Research Approaches	30
2.4.1. Adaptive Workflow Management	30
2.4.2. Cooperative Workflow Management	37
2.4.3. Summary	40
2.5. Artificial Intelligence Approaches	41
2.5.1. Planning	41
2.5.2. Cooperative Agents	44
2.5.3. Temporal Reasoning	47
2.5.4. Summary	48
2.6. Chapter Summary	49

CHAPTER 3	<i>AgentWork Overview</i>	51
3.1.	Layers and Components of AgentWork	51
3.2.	Workflow Definition and Execution	53
3.2.1.	Global Data Schema	53
3.2.2.	Workflow Definitions	55
3.2.3.	Workflow Instances	56
3.3.	Rules for Control Flow Failures	56
3.3.1.	ECA Rules with Control Actions	57
3.3.2.	Composite Events	58
3.3.3.	Activity Patterns	59
3.3.4.	Valid Time	59
3.4.	Control Action Processing	61
3.4.1.	Strategy Selection	61
3.4.2.	Workflow Estimation	65
3.4.3.	Structural Workflow Adaptation	67
3.4.4.	Adaptation Side-Effects	68
3.4.5.	Workflow Monitoring	71
3.5.	Summary and Discussion	72
CHAPTER 4	<i>Data and Rule Definition with ActiveTFL</i>	75
4.1.	Language Requirements	75
4.1.1.	Support of Object-Oriented or Object-Relational Data Models	76
4.1.2.	Support of Rule Definitions	76
4.1.3.	Support of Temporal Aspects	76
4.1.4.	Declarative Semantics	79
4.1.5.	Conclusion	79
4.2.	Frame Logic (F-Logic)	80
4.2.1.	F-Logic Classes and Objects	81
4.2.2.	F-Logic Predicates	86
4.2.3.	F-Logic Formulas	86
4.2.4.	F-Logic Queries	87
4.2.5.	F-Logic Rules	87
4.3.	Temporal Frame Logic	87
4.3.1.	Temporal Frames	88
4.3.2.	Temporal Durations and Distances	88
4.3.3.	Temporal Functions	88
4.3.4.	Temporal Formulas	89
4.3.5.	Conditional Valid Time in ActiveTFL	91

4.4. Active Rules	93
4.4.1. Events	94
4.4.2. Actions	100
4.5. Summary	101
 CHAPTER 5 <i>Workflow Definition and Execution</i>	 103
5.1. Design Goals	103
5.2. Auxiliary Definitions	105
5.2.1. Named Object Patterns	105
5.2.2. Time-Constrained Object Patterns	105
5.3. Workflow Definition Model	106
5.3.1. Basic Activity Definitions	106
5.3.2. Activity Compensation	108
5.3.3. Resource Definitions	108
5.3.4. Condition Definitions	110
5.3.5. Control Flow Definitions	111
5.3.6. Data Flow Definitions	120
5.3.7. Basic Workflow Definitions	125
5.3.8. Complex Activity Definitions	126
5.3.9. Complex Workflow Definitions	126
5.3.10. Workflow Cooperation	127
5.4. Workflow Execution Model	129
5.4.1. Edges States	130
5.4.2. Node States	131
5.4.3. Edge and Node Execution	135
5.4.4. Block Execution	140
5.4.5. Workflow Instance States	141
5.4.6. Execution of Complex Activity Nodes	143
5.5. Related Work	143
5.5.1. Petri Net-Based Workflow Modelling	143
5.5.2. State/Activity-Charts	144
5.6. Summary and Discussion	145
 CHAPTER 6 <i>Workflow Duration Estimation</i>	 147
6.1. Estimation Principles and Definitions	147
6.1.1. Estimation Strategies: Average Case, Worst Case, Best Case	148
6.1.2. Acquisition of Duration Estimation Values	149
6.1.3. Conventions and Definitions	151

6.2. Edge Execution Durations	153
6.2.1. Edges for Internal Data Flow	153
6.2.2. Edges for External Data Flow	153
6.2.3. Unconditional Control Flow and Synchronization Edges	155
6.2.4. Control Flow and Synchronization Edges with Waiting Condition	156
6.2.5. Control Flow Edges with Branching Condition	156
6.2.6. Summary	158
6.3. Node Execution Durations	158
6.3.1. Control Nodes	159
6.3.2. Activity Nodes	159
6.3.3. Communication Nodes	162
6.3.4. Summary	163
6.4. Execution Duration of Sequences and Blocks	163
6.4.1. Activity and Communication Sequences	163
6.4.2. AND-SPLIT/AND-JOIN Blocks	164
6.4.3. OR-SPLIT/OR-JOIN Blocks	165
6.4.4. LOOP-START/LOOP-END Blocks	167
6.4.5. Nested Blocks	168
6.5. Execution Duration of Arbitrary Control Flow Paths	169
6.5.1. Control Flow Paths Without Synchronization	169
6.5.2. Control Flow Paths With Synchronization	171
6.5.3. Entire Workflows	172
6.5.4. Control Flow Path Estimation during Predictive Adaptations	172
6.6. Related Work	173
6.7. Summary and Discussion	175
 CHAPTER 7 <i>Control Actions</i>	 177
7.1. Global Control Actions	178
7.2. Local Control Actions	179
7.2.1. Non-Additive Local Control Actions	180
7.2.2. Additive Local Control Actions	183
7.2.3. Further Aspects	184
7.3. Valid Time Conventions for Control Actions	186
7.4. Control Action Processing	188
7.4.1. Failure Node Set	189
7.4.2. Processing of Global Control Actions	189
7.4.3. Processing of Local Control Actions	191
7.5. Integrity of Failure Rules	197

7.5.1. Rule Redundancy	197
7.5.2. Rule Incompatibility	201
7.5.3. Rule Termination	204
7.6. Dynamic Control Action Dependencies	205
7.6.1. Dynamic Dependencies between Global Control Actions	205
7.6.2. Dynamic Dependencies between Global and Local Control Actions	205
7.6.3. Dynamic Dependencies between Local Control Actions	207
7.7. Summary and Discussion	210
 CHAPTER 8 <i>Structural Adaptation Operators</i>	 211
8.1. Design Goals, Basic Assumptions, and Definitions	211
8.1.1. Design Goals	211
8.1.2. Basic Assumptions	213
8.1.3. Definitions	214
8.2. Control Flow Operators	215
8.2.1. Operator for Node Dropping	216
8.2.2. Operator for Changing Attribute Values	221
8.2.3. Auxiliary Operator for New Parallel Paths	221
8.2.4. Operator for Single Node Adding	228
8.2.5. Operator for Repetitive Node Adding	234
8.2.6. Operator for Replacing Activity Definitions	235
8.2.7. Operator for Node Postponement	241
8.3. Data Flow Adaptation	247
8.3.1. Conditions for Data Flow Adaptation	247
8.3.2. Data Flow Adaptation Strategies	248
8.3.3. Operator for Adding Data Flow Edges	250
8.4. Summary and Discussion	255
 CHAPTER 9 <i>Predictive Control Flow Adaptation</i>	 259
9.1. Transformation of Valid Time Intervals	260
9.2. Algorithm for Predictive Control Flow Adaptation	261
9.2.1. Goals and Principles	261
9.2.2. Details and Illustrating Example	263
9.3. Workflow Monitoring after Predictive Control Flow Adaptation	269
9.3.1. Monitoring Task and Required Meta Information	270
9.3.2. Monitoring for Non-Additive Control Actions	272
9.3.3. Monitoring for Additive Control Actions	273

9.4. Summary and Discussion	274
 CHAPTER 10 <i>Handling Control Flow Failures for Cooperating Workflows</i>	 275
10.1. Temporal and Qualitative Workflow Constraints	278
10.1.1. Temporal Constraints	278
10.1.2. Qualitative Constraints	279
10.2. Handling Global Control Flow Failures for Cooperating Workflows	280
10.2.1. Handling Workflow Abortions for Cooperating Workflows	280
10.2.2. Handling Workflow Suspensions for Cooperating Workflows	281
10.3. Handling Local Control Flow Failures for Cooperating Workflows	282
10.3.1. Determining Temporal Implications	282
10.3.2. Determining Qualitative Implications	283
10.4. Handling of Constraint Violations by Affected Cooperation Partners	284
10.5. Summary and Discussion	284
 CHAPTER 11 <i>Implementation Issues</i>	 285
11.1. Implementation Principles	286
11.2. Workflow Definition and Execution Layer	287
11.2.1. Workflow Editor	287
11.2.2. Workflow Engine	288
11.3. Communication and Integration Layer	291
11.3.1. Principal Structure of CORBA	292
11.3.2. Corba-based Implementation of Communication and Integration Layer	293
11.4. Layer for Handling Control Flow Failures	296
11.4.1. Event Monitoring Agent	296
11.4.2. Adaptation Agent	301
11.5. Summary	306
 CHAPTER 12 <i>Summary and Future Work</i>	 307
12.1. Summary	307
12.2. Future Work	312
 <i>References</i>	 315

Foreword and Acknowledgements

The work described in this dissertation has been performed during my time as a scientific staff member at the Database Section (Head: Prof. Dr. Erhard Rahm) of the Department of Computer Science, University of Leipzig, Germany. The work has been influenced by the HEMATOWORK project [MÜLLER ET AL. 1998] performed in cooperation with the Department for Medical Informatics, Statistics and Epidemiology (Head: Prof. Dr. Markus Löffler), and by practical experiences made during the THEMPO project [MÜLLER ET AL. 1997 B] at the Department of Medical Statistics and Documentation (Head: Prof. Dr. Jörg Michaelis) of the University of Mainz, Germany, where I have been a scientific staff member from 1994-1996.

I am very grateful to Prof. Erhard Rahm for being my advisor and mentor. The scientific guidance and constructive criticism he provided has helped me greatly in conducting my research and developing this dissertation. In many discussions concerning preliminary versions of this dissertation he contributed significantly to the success of this work. In particular, it has been his problem-oriented view, his concentration on the essentials and his accuracy that crucially contributed to my scientific publications, courses, and especially this dissertation.

I wish to sincerely thank Prof. Dr. Peter Dadam (University of Ulm, Germany) and Prof. Dr. Gerhard Brewka (University of Leipzig) for their willingness to be co-reviewers of this dissertation. They both gave me the opportunity to present preliminary results at their departments, and gave substantial remarks which helped to improve the dissertation significantly.

I further want to thank Prof. Dr. Jörg Michaelis, Prof. Dr. Klaus Pommerening, and Dr. Helmut-Matthias Dittrich. During my time in Mainz, they helped me to understand the structure of protocol-directed cancer therapy. This knowledge acquired in Mainz has been an important prerequisite for me to apply workflow technology to hematological treatment, as described in this dissertation.

My special thanks are to Dr. Manfred Reichert (University of Ulm), who deeply went into the material and gave many valuable remarks concerning the conceptual and technical content and the presentation. Furthermore, I want to thank Dr. habil. Frank Wolter for his very helpful remarks on the logic-related parts of the dissertation, Dr. Barbara Heller for her annotations on aspects of knowledge representation, and Prof. Dr. Christian Wolff for making many useful suggestions concerning the first chapters. Dipl.-Inf. Ulrike Greiner did substantial proof-reading and helped me to clarify my thoughts in many technical discussions. Additionally, I want to thank all students who worked together with me in the AGENTWORK and HEMATOWORK project, namely (in chronological order) Frank Brümmer, Enrico Jödecke, Uwe Neubert, Frank Fiebig, Sven Kitschke, Alexander Dietzsch, Rainer Böhme, and Thomas Pippig.

Thanks also to my colleagues of the database section, namely Thomas Stöhr, Timo Böhme, Dieter Sosna, and Andrea Hesse, either for their mental support or their forbearance that I have not always been an uncomplicated colleague especially during the last year when completing this dissertation.

Furthermore, several friends have helped me significantly especially during the last year of writing this dissertation, either by their mental support, appreciation, or practical help, namely Anke and Erik Scheil, Agnes Fabian, Katharina Stengler-Wenzke, Stefan Schmidt, and Thomas Blatt.

Last, but not least, I am indebted to my family for giving me tremendous support, help, and appreciation during this work.

Introduction and Problem Description

Workflow management is widely accepted as a core technology to support long-term business processes in heterogeneous and distributed environments [GEORGAKOPOULOS ET AL. 1995, REICHERT & DADAM 2000, FISCHER 2002]. However, conventional workflow management systems do not provide sufficient flexibility support to cope with the broad range of failure situations that may occur during workflow execution. In particular, most systems do not allow to dynamically adapt a workflow due to a failure situation, e.g., to dynamically drop or insert execution steps [SHETH 1997, REICHERT & DADAM 1998, HORN & JABLONSKI 1998].

As a contribution to overcome these limitations, this thesis introduces the agent-based workflow management system AGENTWORK [MÜLLER & RAHM 1999, MÜLLER & RAHM 2000]. AGENTWORK supports the definition, the execution and, as its main contribution, the event-oriented and semi-automated dynamic adaptation of workflows. AGENTWORK originates from the HEMATOWORK system which addresses workflow support for cancer treatment and which is currently developed at the University of Leipzig [MÜLLER ET AL. 1998]. Though important conceptual decisions are motivated by this medical workflow application, AGENTWORK has been designed to be usable in other workflow application domains as well (such as insurance business or banking).

This introductory chapter is organized as follows: In Section 1.1, we briefly characterize workflow management systems. In Section 1.2, we give a classification of failure situations relevant for workflow management systems. In particular, we introduce so-called *logical failures* and *control flow failures* as an important subclass of logical failures. Informally, a control flow failure occurs if for some reason the control flow of a running workflow is not adequate anymore from the viewpoint of the workflow application. As a consequence, such an inadequate workflow may have to be

aborted, suspended or dynamically adapted (e.g., by dropping, inserting, or replacing execution steps). In Section 1.3, we characterize control flow failures more comprehensively and discuss the requirements that have to be met when dealing with this failure type. Section 1.4 describes the specific contributions of this thesis w.r.t. the handling of control flow failures in the context of workflow management. Section 1.5 briefly describes the HEMATOWORK system to which the concepts and implementation of AGENTWORK have been applied. Section 1.6 completes the chapter with a final overview of the structure of this thesis.

1.1 Workflow Management

For enterprises and organizations operating in global and complex business or public environments, an increased productivity, profitability, flexibility and quality insurance are critical factors for success [REICHERT & DADAM 2000]. A major precondition for this is that organizational structures and business processes are identified, analyzed and optimized continuously in the context of computer-based business process reengineering [HAMMER & STANTON 1995, SCHEER 1998 A]. In this connection, the term *business process* commonly refers to a logical unit of work relevant for an enterprise or an organization. Typical examples for business processes include the processing of a bank customer's credit application, the processing of a damage case at an insurance company, or the administration of a chemotherapy to a cancer patient. More formally, a business process is a set of business *activities* and their *temporal relationships*. For example, a typical activity is to check whether an insurance holder has filled out a car damage report correctly, or the performing of a x-ray examination in a hospital. The temporal relationships specify which activities have to be processed sequentially, in parallel, or only when certain conditions hold. Business processes may also be distributed over multiple locations due to enterprise collaborations.

Traditionally, information systems have been implemented in a function- and data-centered manner [BREITBART ET AL. 1993], and do not have any explicit notion of business processes. This implies that users themselves have to know when an execution step has to be performed. In particular, the *passive* behavior of such traditional information systems does not support any work coordination, and does not provide mechanisms for process monitoring such as generating reminders in case of deadline violations. Consequently, *process-oriented* information systems are required that *actively* provide the right data together with the right execution instructions to the right staff member at the right time.

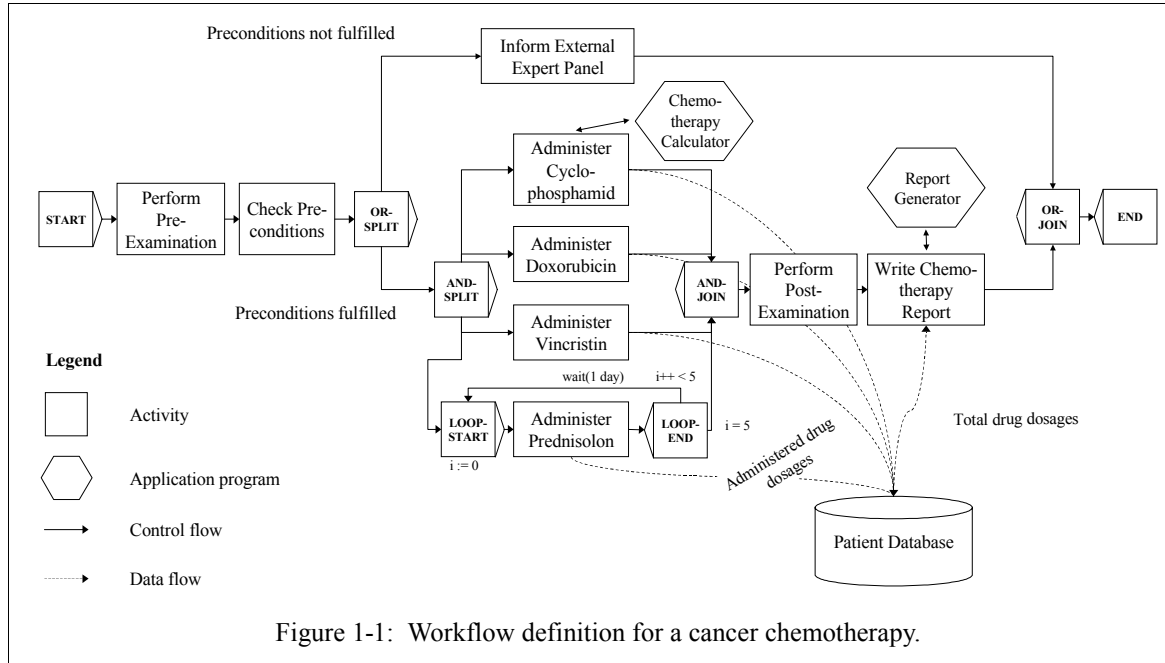
Conventionally, such process-oriented information systems have been implemented by incorporating the process logic directly into the application programs. This means that the different application programs supporting activities of the business process invoke each other according to the process logic, and communicate bilaterally. Though this allows to actively operate a business process, it has the serious disadvantage that the overall execution logic of a business process is hidden and split up within the code of the different application programs. As a consequence, tasks that go beyond the scope of a single application program cannot be supported significantly, as a higher-level control component is missing. For example, if such a higher-level control component is missing, it is difficult to determine which processes are at which execution steps, which users work on

which documents and so on. Furthermore, failure handling such as setting a running process into a consistent state when an application program or database server crashes is nearly impossible. Additionally, process maintenance is difficult as any change of the process logic has to be realized by changing the code of the different application programs. For a detailed discussion on the disadvantages of passive information systems and conventional process-oriented information systems, we refer to [REICHERT 2000].

To overcome these limitations, *workflow management systems* have been introduced [GEORGAKOPOULOS ET AL. 1995, ALONSO & MOHAN 1997, LEYMANN & ROLLER 2000]. Their basic characteristic is that they strictly separate the application program code from the overall logic of a business process. This allows to support a broad range of tasks going beyond the scope of a single application program, such as process monitoring, failure handling and load balancing in the sense of assigning execution steps to free human or machine resources. In workflow management systems, the overall process logic is explicitly represented in an executable *process* or *workflow definition*. Such a workflow definition first consists of a *control flow* definition specifying in which order the activities have to be executed. Second, it consists of a *data flow* definition specifying which data is needed as input for the activities, and which data is produced as output of the activities. Furthermore, the data flow specifies when data has to be retrieved from or written to data sources such as relational databases or user interfaces. Third, the workflow specifies which *workflow resources* (e.g., users, application programs, equipment) are needed for the execution of an activity. In addition to such *intra-workflow* aspects, some workflow management systems also allow to specify *inter-workflow* aspects which are needed for workflow cooperation. For example, such an inter-workflow specification may define when one workflow has to provide a result for another (remote) workflow, and which temporal or qualitative constraints have to be met by this result. Orthogonal to this, workflows may also be organized *hierarchically*, i.e., to an activity a *sub-workflow* may be assigned which is executed when the control flow reaches this activity.

Figure 1-1 gives an example of a workflow definition for the domain of cancer therapy. This workflow definition specifies that first a pre-examination has to be performed for the patient to which this chemotherapy shall be administered (e.g., it is checked whether the patient has an infection which forbids to administer the chemotherapy drugs). If the preconditions are met, the four drugs of the chemotherapy (i.e., CYCLOPHOSPHAMID, DOXORUBICIN, VINCRISTIN, and PREDNISOLON) are administered. In particular, PREDNISOLON is given five times. Then, a post-examination is performed, and a final chemotherapy report is written. If the preconditions for the chemotherapy are not met, an external expert panel is informed which gives advice to the treating physicians how to cope with this patient. Furthermore, the workflow definition of Figure 1-1 specifies that two application programs – namely a chemotherapy calculator determining the patient-specific drug dosages and a report generator – are required for the execution of some activities. Concerning the data flow, the definition of Figure 1-1 specifies that the administered drug dosages have to be stored in the patient database, and that the total drug dosages of the entire chemotherapy are needed as input for the *Write Chemotherapy Report* activity.

Figure 1-2 shows the core components and key aspects of a typical workflow management system, as suggested by the WORKFLOW MANAGEMENT COALITION [FISCHER 2002, WPMC 2002]:



Workflow Definition Tool: This tool supports the definition of workflow activities and of the control and data flow between these activities. In particular, references to the organization model can be specified to express which activities have to be executed by which organizational units and staff members. Analogously, references to application programs can be specified to indicate which programs are required for which activities. Depending on the particular workflow management system, the definition of the organizational model and the applications programs itself either can be done by the workflow definition tool, or can be imported from other tools (not shown in Figure 1-2). Typical workflow definition tools allow to define workflows with a graph-based language (such as the one shown in Figure 1-1). Sometimes also a verification component is provided to avoid incomplete or inconsistent workflow definitions.

Workflow Enactment System: The *workflow enactment system* is responsible for the execution of workflows. Its core component is the so-called *workflow engine*. The workflow engine instantiates workflows from workflow definitions, and decides which activities of the workflow have to be executed next. To control and monitor workflow execution and to handle failure situations, the workflow engine maintains so-called *workflow control data*. For example, these workflow control data describe the actual execution stage of a workflow and its activities, or record the execution chronology. In particular, workflow control data cannot be manipulated by application programs or users. During workflow execution, the workflow engine invokes application programs if they have been assigned to the activity to be executed next. For activities which have to be executed by users, a *worklist handler* maintains the worklists stating which activities are to be executed by which staff

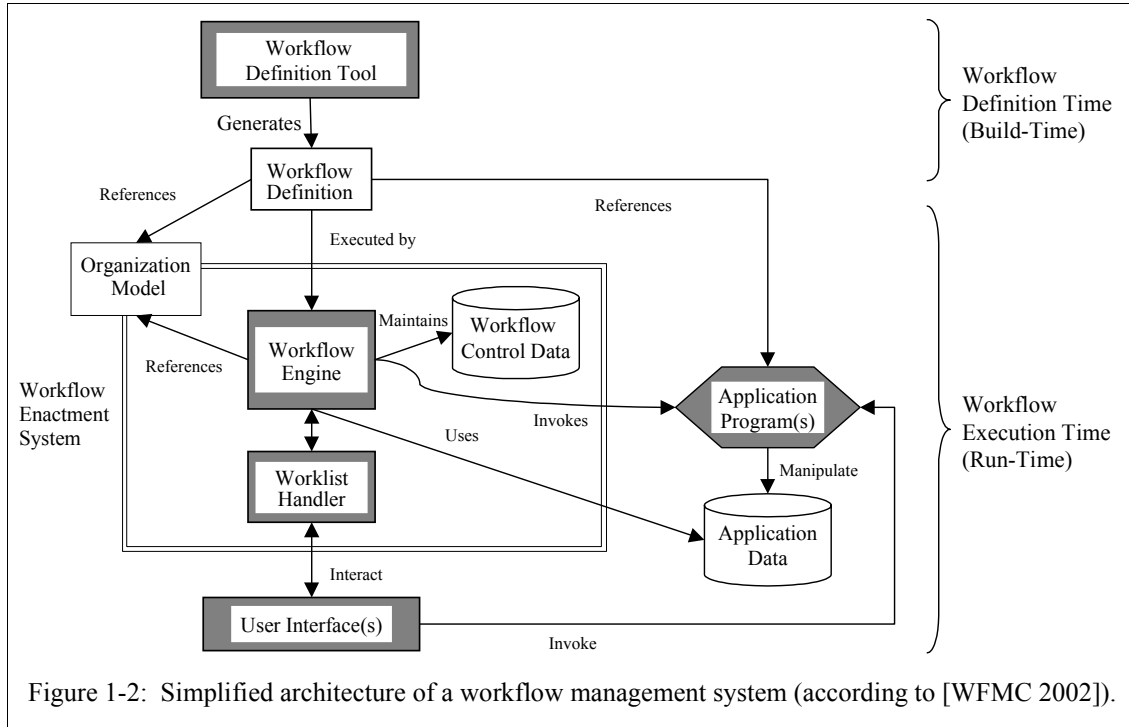


Figure 1-2: Simplified architecture of a workflow management system (according to [WFMC 2002]).

members, and propagates this information to the respective user interfaces. From these user interfaces, application programs may also be invoked (instead of being invoked by the workflow engine itself). These application programs maintain *application data*, such as data about patients or customers. In contrast to workflow control data, application data may be also maintained directly by users. Some application data may also be used by the workflow engine itself, e.g., to evaluate a conditional branching such as the "Preconditions fulfilled" branching of Figure 1-1.

For further architectural aspects of workflow management systems, we refer to [BAUER & DADAM 2000, GREFEN & REMMERTS DE VRIES 1998, MILLER ET AL. 1998, MUTH ET AL. 1998 A, JABLONSKI 1997, REINWALD 1993].

1.2 Failure Handling in Workflow Management Systems

A workflow environment typically covers a heterogeneous and distributed collection of different hosts, network components, databases, application programs, and user interfaces. Within this environment, the workflow engine executes workflows, invokes programs and reads or writes data from or to user interfaces, programs and databases. In this context, a broad range of failure situations may occur which have to be handled. Such failure situations can be classified into *device failures*, *system failures*, *transaction failures*, and *logical failures* [HÄRDER & RAHM 2001]. For the latter failure class, so-called *control flow failures* form an important subclass. To better understand

these control flow failures which form the main topic of this thesis, we briefly characterize these principal failure types and the mechanisms to cope with them.

1.2.1 Device, System, and Transaction Failures

Device failures mainly concern disk failures caused, for example, by a head crash. As a consequence, database files containing important information such as workflow definitions or status data of running workflows may be destroyed. *System failures* cover failures in the execution environment of workflows, e.g., malfunctions and crashes of operating system components or database servers. So-called *transaction failures* occur when the workflow system itself or an application program performs invalid operations such as the division by zero or a database tuple insert violating data integrity constraints.

In “classical” database applications device, system and transaction failures usually are handled by transaction management which is based on the so-called ACID¹ paradigm and which is realized by archiving, logging and recovery mechanisms [HÄRDER & RAHM 2001, GRAY & REUTER 1993]. Principally, a transaction is a collection of database operations with the following four basic properties:

1. *Atomicity*: A transaction is either processed entirely or not at all (“all-or-nothing” principle). In case of a failure the database system sets back (“rolls back”) all database operations already performed by the transaction (*undo recovery*).
2. *Consistency*: A transaction transfers a database from one consistent state to another consistent state. Typically, “consistency” is expressed by integrity constraints assigned to database objects. If an integrity constraint is violated at the end of a transaction, the transaction is rolled back.
3. *Isolation*: If a transaction T accesses a database object this object is isolated from other transactions until T is finished. This avoids that other transactions read or update the data processed by T before T has been able to complete its data processing. Most database systems implement isolation by *locking* an object which is accessed by a transaction, and by releasing the lock at the end of the lock-holding transaction.
4. *Durability*: Whenever a transaction has committed, the *durability* of its effects is guaranteed. This means, that database changes performed by a transaction survive all errors occurring after the commit of this transaction, including operating system crashes or device failures. To achieve this, a *redo recovery* may have to be performed by the database system.

However, it has been widely recognized that the classical ACID model is not appropriate to handle device, system or transaction failures for workflows [ALONSO ET AL. 1994, ELMARGARMID 1992]. This is because the ACID model assumes a *short* life span of operations, i.e., that relevant operations typically last only seconds or minutes. This assumption is not valid anymore for

1. ACID = Atomicity, Consistency, Isolation, Durability

workflows. Workflows typically represent long-term processes with an average life time of hours, days, weeks or even months. Therefore, for many workflow applications a rollback of a workflow is not acceptable as this would imply that too much work already done by the workflow would be lost. Additionally, workflow activities such as administering a drug infusion cannot simply be rolled back *at all*. Furthermore, strict isolation of a database object for the whole life span of a workflow often would mean that other workflows needing this object as well would have to wait unacceptably long for this object.

Consequently, during the last years there have been many efforts to generalize and relax the classical ACID model to achieve a more workflow-oriented transaction management [RUSINKIEWICZ & BREGOLIN 1997, JAJODIA & KERSCHBERG 1997, GEORGAKOPOULOS & HORNICK 1994]. For example, to reduce the loss of work as much as possible, the so-called *forward recovery* model [WÄCHTER & REUTER 1992] does not roll back a running workflow entirely when a failure occurs but *interrupts* it with a minimum of undo recovery (i.e., usually only activities executed at the moment of the failure are rolled back). When the failure has been resolved the workflow is continued on the basis of the already successfully committed activities.

Though this thesis does not focus on workflow transaction management, we will discuss several advanced transaction models in Chapter 2 (*Related Work*) in detail. This is because advanced transaction models such as *compensation-based* transaction models [GREFEN ET AL. 1999 B, LEYMANN 1995] also consider aspects being relevant for the handling of so-called *logical failures*.

1.2.2 Logical Failures

Logical failures occur if for some reason the workflow definition on which a running workflow is based becomes inadequate. For example, imagine that the workflow of Figure 1-1 is executed for cancer patient Bob Miller, and that the loop administering the drug PREDNISOLON has already been started. Suddenly, Bob Miller shows allergic reactions against PREDNISOLON, which could not have been predicted before the loop execution. Therefore, if the control flow that still has to be executed contains activity executions administering PREDNISOLON, these activity executions dynamically have to be removed from the workflow to avoid further allergic reactions of patient Bob Miller. In contrast to this, the execution of the other activity nodes (e.g., the VINCRISTIN node) can be continued without change. In particular, the alternative to *abort the whole workflow* is unacceptable as only a few activity executions (i.e., the PREDNISOLON administrations) are affected by the allergy. Furthermore, other workflows based on the same workflow definition but running for other patients can be executed entirely according to the original workflow definition.

Logical failures may not only cover the inadequacy of control flows as illustrated in the PREDNISOLON example, but also the inadequacy of data flow definitions or temporal constraints (such as deadlines) assigned to a workflow. For example, if it turns out during workflow execution that a deadline assigned to this workflow cannot be met anymore, this deadline may have to be adapted, e.g., extended.

Typically, logical failures are caused by application *events*, e.g., events related to the customers, products and employees of an enterprise, or events concerning the patients, diseases, treatment pro-

cedures and doctors in a hospital. In the example above, the logical failure that the PREDNISOLON activity executions become inadequate for Bob Miller is caused by the application event that an allergy against PREDNISOLON has been detected for this patient. Application events are described by application data introduced in Section 1.1.

A logical failure may be caused not only by events occurring to those entities for which a workflow is executed (e.g., the patients or customers), but also by events occurring to a workflow *resource*, such as a user or piece of equipment. For example, let us assume that a medical workflow consists of a difficult medical activity that requires a lot of experience and therefore can only be performed by a senior physician. If this senior physician becomes ill for a few days and if no adequate proxy is available for this activity, the workflow may also have to be adapted. For example, the corresponding activity could be postponed until the senior physician is back to work while other activities for which the senior physician is not necessary may still be processed as originally planned.

1.2.3 Control Flow Failures

In this thesis, we focus on logical failures characterized by a *dynamic inadequacy of control flow* and caused by application events, such as the inadequacy of the PREDNISOLON activity executions due to an allergy event in the cancer workflow example above. Thus, we (informally) define as follows:

Definition 1.1: Control Flow Failure

A *control flow failure* occurs if an application event has the consequence that – from the application point of view – at least one running workflow becomes inadequate w.r.t. its control flow.

An application event inducing a control flow failure is called a *control flow failure event* or simply a *failure event*.

We focus on *control flow* inadequacy as dealing with this type of logical failures is very important for many application domains but still a major open research problem in workflow management, as we will show in Chapter 2 (*Related Work*). Data flow aspects are only considered when the handling of a control flow inadequacy requires that the data flow has to be handled as well. For example, because of an additional activity execution additional data flow edges may have to be provided as well. In contrast to this, we do not address dynamic deadline adjustment, as this topic has been largely investigated in the context of workflow deadline management and scheduling systems [DADAM ET AL. 2000, BLAZEWICZ ET AL. 2001, EDER ET AL. 1999 A].

We distinguish two main types of control flow failures:

- *Global Control Flow Failures*: In this case an application event implies that a running workflow becomes inadequate w.r.t. its whole remaining control flow (i.e., w.r.t. *all* of its activities). Therefore, the workflow has to be suspended for some time or even has to be aborted. For example, a patient suffering from an unexpected infection during a cancer chemotherapy must

not get cancer drugs anymore. Therefore, the entire chemotherapy workflow has to be suspended until the infection is over. A non-medical example for such a global control flow failure could be that a customer cancels his order. As a consequence workflows processing the customer's order usually do not make much sense anymore and have to be aborted.

Global control flow failures are already covered by several recent workflow failure handling approaches [CASATI 1998, HAGEN & ALONSO 1998]. Therefore, this thesis does *not* concentrate on this control flow failure type. However, it is nevertheless integrated in the failure handling approach of AGENTWORK to achieve completeness.

- *Local Control Flow Failures*: In this case an application event implies that a running workflow becomes inadequate not as a whole but only *locally*, i.e., only some of its activities are not appropriate anymore (such as the PREDNISOLON activities in the cancer example above). The principal mechanism to deal with this type of control flow failures is the following: First, the control flow is dynamically adapted (e.g., by removing or inserting activities). Second, if necessary the data flow is adjusted. Third, the adapted workflow is continued. At any moment during these steps a user interaction must be possible.

Although this type of control flow failure has also been investigated by several authors [CASATI 1998, REICHERT & DADAM 1998, WESKE ET AL. 1998, HEINL ET AL. 1999, BORGIDA & MURATA 1999], there is still a number of serious problems that have not been addressed sufficiently, such as the automated detection and predictive handling of local control flow failures. Therefore, the main focus of this thesis is on this second type of control flow failures. In Section 1.3, we discuss several of these open problems.

It could be argued that one could also cope with control flow failures by adding *conditional branches* to a workflow definition (such as the two branches *Patient has allergy concerning PREDNISOLON* versus *Patient has no allergy concerning PREDNISOLON*). This would avoid the dynamic adaptation of a workflow as the possibility of an allergy concerning PREDNISOLON is already considered in the workflow definition. However, this approach is not appropriate because of the following reasons:

- First, considering control flow failures by conditional branches would lead to an enormous number of different branches within the workflow definitions and would reduce workflow readability and maintenance significantly. For example, during a typical chemotherapy with about 4 drugs every drug may cause about 5-10 different side effects so that about 20-40 conditional branches would be necessary.
- Second, and more important, the failure events causing control flow failures typically occur *asynchronously at any time* during workflow execution, i.e., the relative point in time of their occurrence w.r.t. a particular position in a workflow definition is not known in advance. Considering a failure event by a conditional branch would require that it is known, for example, after which activities this event will occur so that the conditional branch can be placed there. However, this is not possible for most types of failure events.

In artificial intelligence, the problem that for many application classes it is impossible to pre-model all situations is also known as the *qualification problem* [REITER 2001].

We complete this section with

Definition 1.2: Control Action

An action which has to be performed from the application point of view to cope with a control flow failure, is called a *control action*.

In case of a *global* control flow failure, a control action is performed on the workflow level (e.g., *abort* or *suspend* workflow of type *W*). In case of a *local* control flow failure, a control action is performed on the activity level (e.g., *drop* or *replace* activity of type *A*). The formalization and handling of these control actions will be one of the main topics of this thesis.

1.3 Requirements of Handling Control Flow Failures for Workflow Systems

Many current approaches assume that the workflow administrator or an authorized workflow user decides which events constitute control flow failures and which control actions and structural adaptations have to be done w.r.t. running workflows when a control flow failure occurs [WESKE 1999 B, HEINL ET AL. 1999, REICHERT & DADAM 1998]. This manual process can be supported, for example, by a workflow editor which in case of a local control flow failure only allows adaptations leading to a consistent workflow after the dynamic change.

However, in many domains events occur continuously and simultaneously, and typically several workflows run in parallel. Therefore, in such domains a purely *manual* handling of control flow failures is likely to overcharge the user. For example, during a cancer chemotherapy a physician is often faced with 10-30 findings and laboratory values per patient every day. Furthermore, for every patient several workflows usually run in parallel. One may be dealing with diagnostic procedures, another may coordinate the therapeutic procedures. Further supportive workflows may manage the appointments and requests with other involved departments for this patient (such as the radiological department or the central drug store of the hospital). With a manual failure handling, important events constituting control flow failures may be overseen or detected too late. As a consequence workflows would be continued although not being adequate anymore. At least, the manual handling must be viewed as a very time-consuming process in data-intensive domains.

The potential of automating the handling of control flow failures in the workflow context has already been identified by several authors [SINGH & HUHN 1994, DELLEN ET AL. 1997, CASATI 1998, BECKSTEIN & KLAUSNER 1999A]. However, there are still major requirements in this context that are not sufficiently met by current approaches. This includes

1. the representation of failure events and control actions,
2. the translation of control actions into workflow execution operations and structural adaptations,

3. the handling of inter-workflow implications of control flow failures, and
4. the handling of data and event distribution and heterogeneity.

We discuss these requirements in the following to make the rationale of AGENTWORK more transparent and to better motivate the thesis contributions listed in Section 1.4.

Requirement 1: Representation of Failure Events and Control Actions

First of all, events and control actions have to be described on the application level so that they are meaningful for the workflow user such as the physician. This is necessary as it cannot be assumed that the handling of control flow failures can be automated entirely. Thus, the necessity of human interaction requires the representation of events and actions on a high semantic level.

Furthermore, events and control actions in the context of control flow failures may have an arbitrarily complex structure. For example, an event often constitutes a control flow failure only together with other events. More precisely, the question whether an event constitutes a control flow failure may depend on specific *temporal* relationships w.r.t. other events. Concerning control actions, until now we simply stated that in case of a failure event some *actions on workflows or activities* (such as *abort*, *drop* or *add*) have to be performed. Similar to events, the structure of such actions can become arbitrarily complex as well. This holds especially for *local* control flow failures. We give examples for such complex events and control actions from two domains:

- *Domain of Cancer Treatment:* Besides allergies, a typical event that may require to dynamically remove particular drug administrations from a workflow is that the leukocyte count of a patient suddenly becomes significantly low as a drug side-effect (e.g., lower than 1000 per mm³ blood). However, in clinical practice such a low leukocyte value often will be tolerated for several days without adapting the treatment workflow. Only if similar leukocyte counts (e.g., all less than 1000) occur during a time period longer than 4 days this is a serious finding showing a severe blood disorder of the patient and thus requiring a workflow adaptation. Therefore, these similar leukocyte counts *together* (and *only* together) constitute a failure event as a running workflow may have to be adapted to cope with this critical situation.

A simple control action then could be to drop from a treatment workflow *all* activities administering cytostatic drugs which are known to cause a leukocyte cell reduction as a negative side effect. However, a physician would often drop these drug activities *only for the next 7 days*, or *until* the leukocyte count again is higher than 2500². This is because the blood system usually recovers during this time so that blood-toxic cytostatic drugs can again be administered after this. Alternatively, the physician could also drop these drug activities until the leukocyte count is higher than 2500/mm³ for more than 3 days.

- *Domain of Insurance Business:* An insurance holder reports by telephone that he caused a minor car accident with an assessed damage sum of 700\$ w.r.t. the damaged car. Furthermore, he reports that his velocity was about 30 miles/h. At the insurance office a workflow is started

2. During chemotherapy, leukocyte counts higher than 2500 indicate an acceptable, non-critical range.

to deal with the damage. As the damage is a minor one the respective workflow definition does not contain a technical expert report. However, two days later the driver whose car was damaged by the insurance holder sends the requested accident report. There he states that the velocity of the car causing the accident was at least 45 miles/h. This is inconsistent to the previous velocity statement of the insurance holder. Both velocity reporting events together therefore induce a control flow failure. A simple control action then could consist of dynamically adding a technical expert report activity into the workflow to clarify the situation.

A more complex control action is described in the following example: An insurance workflow deals with a customer's proposal for a car liability insurance. After the proposal acceptance the workflow sends the insurance contract and a bill for the first fee to the customer. If the customer does not pay the fee within 2 weeks the workflow usually sends a payment reminder to the customer, and a second one after another week. If the customer does not respond to the second reminder within one week, the insurance contract is retrospectively declared inoperative and cancelled. Let us now assume that for a particular customer such a first payment reminder is prepared by the workflow. However, this customer is currently also in negotiations with the enterprise w.r.t. an expensive life insurance. This constitutes a control flow failure inducing the following control action: To avoid a "disturbance" of the life insurance negotiations by payment reminders, all reminder activities are dropped from the workflow *as long as this customer has not signed the life insurance contract offered to him, or until additional 2 weeks have been exceeded w.r.t. the payment of the car insurance fee.*

The complexity of events and actions illustrated in these examples is not uncommon in many application domains and therefore cannot be neglected in the context of workflow failure handling. The representation of events, actions and their interdependencies has already been addressed in the fields of active and deductive databases [DAYAL ET AL. 1996, WIDOM & CERI 1996, PATON 1999, LUDÄSCHER 1998] and artificial intelligence [GIARRATANO & RILEY 1993, BUCHANAN & SHORTLIFFE 1984]. However, especially the *temporal* dimension of failure events and of the induced control actions still is a major problem that has to be addressed in the future.

Requirement 2: Translation of Control Actions into Workflow Execution Operations and Structural Adaptations

If a control action such as "*abort entire workflow*" or "*drop cytostatic drug activities for the next 7 days*" has been derived because of some failure event this has to be translated into appropriate operations on the workflow execution level.

In case of a workflow *abortion*, a running workflow is not something that simply can be "switched off". Active programs and data processes triggered by the running workflow may have to be terminated carefully, and the effects of the workflow may have to be made undone or at least compensated. In case of a *suspension* the workflow does not have to be aborted but may have to be interrupted at the current execution state, and continued later on. As workflow abortion and interruption has already been addressed largely by advanced workflow-oriented transaction models [WÄCHTER & REUTER 1992, ALONSO ET AL. 1996, WORAH & SHETH 1997, GREFFEN ET AL. 1999 B]

we do not focus on these topics in this thesis.

In case of *local* control flow failures a control action such as “*replace activity of type A by activity of type B during the next 7 days*” has to be translated into appropriate structural adaptations of running workflows. For example, the “*during the next 7 days*” constraint above implies that from a running workflow all activity nodes of type *A* that are going to be executed *during the next 7 days* have to be replaced by activity nodes of type *B*. Principally, two main strategies can be identified to deal with this problem:

Strategy 1: Reactive Adaptation

A straightforward strategy consists of monitoring the workflow for the specified temporal interval and of determining for every activity to be executed whether it is affected by the control flow failure. For example, if a particular drug has to be dropped for the next seven days, then the workflow is monitored for this interval, and whenever the control flow reaches a node administering this drug, this node is skipped or dropped.

This strategy is simple to realize and has the advantage that at the moment of the failure event one does not have to know when an activity node will be executed during some temporal interval, such as during “*the next 7 days*” interval of the “*replace activity of type A ...*” example above. However, reactive adaptation has the serious limitation that it may be *too late*. This is because the dropping or adding of activities often requires a comprehensive preparation. For example, the dropping of an expensive cytostatic infusion requires that – if possible – the central drug store is informed two days before so that the infusion is not unnecessarily prepared. Otherwise, the infusion has to be poured away as it cannot simply be administered to other patients because of the patient-specific concentrations. It could be argued that such preceding or preparing activities could also be derived during an automated failure handling process, and therefore could automatically be added to a running workflow by strategy 1 above. However, this would require that *all* activity types of an organization are represented electronically within the workflow system. This must be viewed as unrealistic especially w.r.t. the first phases of bringing a workflow system into practice. Therefore, many local control flow failures require that necessary structural adaptations are known and applied as soon as possible so that the staff has enough time for preparing steps not known to the workflow system.

Strategy 2: Predictive Adaptation

An alternative strategy to translate a local control action into structural workflow adaptations consists of predictively determining which parts of the control flow of an affected workflow correspond to the temporal interval assigned to this control action. More technically, this means that this strategy makes an estimation which parts of a workflow’s control and data flow *will be executed in the future during this temporal interval* (e.g., will be executed during the next 7 days w.r.t. the *replace A by B* example above). If such a part corresponding to this temporal interval has been identified, structural adaptation operators have to be applied which, for example, remove or add nodes and which also remove, generate or rearrange edges connected with these nodes. This gives the staff the possibility to perform preparing steps which are – for some reason – not known to the

workflow system.

Principally, this strategy usually models human failure handling better than the non-predictive strategy 1 above. However, strategy 2 has two limitations. First, the temporal estimation requires meta knowledge about the duration of activities. This limitation can be overcome as in many domains at least heuristic information about the average duration of activities is known. Second, and more serious, such an estimation may become imprecise or even impossible if the control flow contains, for example, OR splits or loops. Furthermore, subsequent adaptations may make former estimations invalid, so that former adaptations may have to be reevaluated.

Therefore, the main requirement concerning the translation of control actions into workflow execution operations and structural adaptations consists of providing mechanisms to decide whether reactive or predictive adaptation – or a combination of them – is appropriate to handle a local control action. This decision will depend on the structure of the affected workflow and the type of triggered control action. Furthermore, the *consistency* of an adapted workflow and the *efficiency* of the adaptation has to be guaranteed.

Concerning this requirement 2, which we can describe in more abstract terms as the problem of automatically translating *declarative* control action statements into appropriate adaptations of *procedural* constructs, some work has already been done in the field of artificial intelligence planning [MYERS 1998, HAMMOND 1990]. However, the problem has not yet been addressed sufficiently in the context of workflow management. This will be discussed in detail in Chapter 2 (*Related Work*).

Requirement 3: Handling of Inter-Workflow Implications of Control Flow Failures

As mentioned in Section 1.1, a workflow definition may consist of inter-workflow aspects such as workflow cooperation, e.g., the specification when one workflow has to provide a result for another (remote) workflow, and which temporal or qualitative constraints have to be met by this result. Thus, an important problem in the context of handling control flow failures is the question how an abortion, suspension or dynamic adaptation of a workflow affects *other workflows* cooperating with this workflow. A workflow abortion, for example, may cause that a result cannot be provided anymore for a cooperating workflow, so that this cooperating workflow should be informed.

Again the situation becomes more complicated when *local* control flow failures occur and therefore workflows are *adapted*. Then, a dynamic deletion or insertion of activities w.r.t. the workflow providing the result may imply that a cooperating workflow still will receive the result, but probably later or not in the quality on which both originally agreed. We give motivating examples for inter-workflow implications of local control flow failures from two domains:

Domain of Cooperative Medical Care: In many medical disciplines patient treatment is performed not only by a single hospital ward or division, but by several cooperating departments. For example, during the treatment of a cancer patient the departments for internal medicine and radiotherapy closely cooperate. A workflow system at the department of internal medicine may support the physicians w.r.t. the chemotherapy of a patient. Another workflow system at the radiological department may support tasks such as the preparation, performance and aftercare of radiotherapy procedures. A typical treatment could consist of a two weeks chemotherapy at the department of

internal medicine and parallel units of supportive radiotherapy every two days (for radiotherapy, the patient has ambulant appointments at the radiological department). If a control flow failure such as an unexpected allergy w.r.t. the drug CYCLOPHOSPHAMID occurs, this may require dynamic adaptations of the chemotherapy workflow such as deleting the activities administering CYCLOPHOSPHAMID. This adaptation may impact the radiotherapy workflow. As the dropped drug is essential for tumor reduction, additional radiological units may become necessary to compensate the cancellation of this drug. Thus the deletion of activities from one workflow can make it necessary to insert additional activities into a cooperating workflow.

Domain of Banking (Credit Check): When an industrial customer company applies to a bank for a high project credit, the responsible *Credit Management* group involves the division *Financial Securities* to judge the financial securities of the customer and the division *Project Evaluation* to evaluate the success perspectives of the project. For this purpose, both divisions have their own workflow system. Furthermore, the *Credit Management* group has a workflow system for the coordination of the divisions and for the final decision process. A control flow failure at *Financial Securities* could be that a real estate assessment of an external expert turns out to be incomplete or wrong. Therefore, additional activities may have to be inserted to provide the necessary information (by instructing, for example, a second assessment expert). In this case, the *Credit Management* group is affected as the credit notification date which was originally fixed with the customer may not be realistic anymore. At the *Project Evaluation* division, a control flow failure may be that the customer suddenly drops a risky part of the project. As a consequence an activity performing an evaluation of high-risk subprojects may be dynamically deleted. As a consequence, activities for high-risk decisions may also be deleted from workflows of the *Credit Management* group, so that the customer can be informed earlier.

The problem that control flow failures of a workflow may also impact cooperating workflows is an additional argument for the predictive adaptation strategy discussed above: If adaptations are performed predictively, cooperation partners can be informed timely and can prepare themselves according to the new situation. Concerning the example of cooperative medical care above, the radiological staff would then have enough time to prepare additional radiotherapy units if required.

Cooperation aspects of distributed workflow systems have already been addressed by several authors. In particular, aspects such as workflow interoperability frameworks [ADAMS & DWORKIN 1996, BUSSLER 1998], workflow synchronization [HEINLEIN 2001, ATTIE ET AL. 1996] and collaboration management infrastructures [BAKER ET AL. 1999] have been investigated. During the last years, workflow cooperation has received significant additional attention because of the strongly evolving fields of electronic commerce and virtual enterprises [ALONSO ET AL. 1999, NGU 1999]. However, not much work has been done yet to cope with failure handling in inter-workflow cooperation scenarios. In particular, the question how to automatically determine which cooperating workflows are affected by the dynamic adaptation of a workflow and how these cooperating workflows have to be informed or adapted as well has not been investigated sufficiently. We will discuss these aspects of related work in Chapter 2, too.

Requirement 4: Handling of Data and Event Distribution and Heterogeneity

Until now, when talking about events and control flow failures we ignored aspects such as the location and format of the application data representing these events. However, this data is usually stored in a distributed and heterogeneous environment. Therefore, as an important additional requirement, a workflow system dealing with control flow failures first must be able to *notice* events occurring somewhere in a distributed and heterogeneous environment. Second, it must be able to deal with the heterogeneity of the data representing these events.

1.4 Contributions of the Thesis

Based on the requirements listed above we can now describe the contributions of this thesis more precisely. Principally, the overall contribution of this thesis is the design and prototypical implementation of the agent-based workflow management system AGENTWORK. Like other workflow management systems, AGENTWORK provides components for the definition and execution of workflows. In contrast to most other systems, it provides *additional* mechanisms to semi-automatically handle control flow failures. We emphasize that the AGENTWORK approach is mainly motivated by medical application domains, but is assumed to be useful also for other non-trivial domains. This assumption is based on the observation that the data and process complexity of medicine subsumes that of many other application domains [DADAM ET AL. 2000].

The following contributions are described in this thesis. Contribution 1 does not specifically address one of the requirements listed above, but forms an important and necessary prerequisite.

1. ***A Formal Model for Workflow Definition, Execution, and Estimation:*** In this context, AGENTWORK first provides a workflow definition language. This language allows for the definition of a workflow's control and data flow. Furthermore, a workflow's communication and cooperation with other workflows or workflow systems can be specified. In particular, the workflow definition language supports an object-oriented data flow model. Such a model has been considered necessary to cope with the data complexity of domains such as medicine. Second, AGENTWORK provides a precise workflow execution model. This is necessary, as a running workflow usually is a complex collection of concurrent activities and data flow processes. Therefore, it has precisely to be defined what terms such as *workflow execution*, *abortion* or *suspension* mean in the context of handling control flow failures.

Furthermore, mechanisms for the estimation of a workflow's future execution behavior are provided. These mechanisms are of particular importance for predictive adaptation.

This is described in Chapter 5 (*Workflow Definition and Execution*) and Chapter 6 (*Workflow Duration Estimation*).

2. ***Mechanisms for Determining and Processing Failure Events and Control Actions*** (Contribution to requirement 1 in Section 1.3): AGENTWORK provides mechanisms to decide whether an event constitutes a control flow failure and which control actions have to be performed to cope with this failure. In particular, these mechanisms decide which workflows are affected by con-

trol actions and therefore may have to be aborted, suspended, or adapted. This is formally done by evaluating event-condition-action rules where the event-condition part describes under which condition an event has to be viewed as a failure event. The action part represents the necessary control action. To overcome the limitations of current event-condition-action approaches especially w.r.t. the temporal dimension of events and actions, the thesis provides a novel event-condition-action model based on a temporal object-oriented logic. For this purpose an existing object-oriented logic (FRAME LOGIC [KIFER ET AL. 1995]) has been extended with temporal operators to better deal with the time-oriented aspects of events and actions. Furthermore, an approach is introduced which supports the integrity of event-condition-action rule sets.

This is described in Chapter 7 (*Control Actions*).

3. ***Mechanisms for the Adaptation of Affected Workflows*** (Contribution to requirement 2 in Section 1.3): In case of *local* control flow failures it has to be decided how an affected workflow has to be dynamically adapted. AGENTWORK provides a novel approach that combines the two principal strategies reactive adaptation and predictive adaptation discussed w.r.t. requirement 2 in Section 1.3. Depending on the context of the failure, the appropriate strategy is selected. Furthermore, *control flow* adaptation operators are provided which translate local control actions into structural control flow adaptations. *Data flow* operators adapt the data flow after a control flow adaptation, if necessary.

These aspects are described in Chapter 8 (*Structural Adaptation Operators*) and Chapter 9 (*Predictive Control Flow Adaptation*).

4. ***Mechanisms for the Handling of Inter-Workflow Implications of Control Flow Failures*** (Contribution to requirement 3 in Section 1.3): AGENTWORK provides novel mechanisms to decide whether a control flow failure occurring to a workflow affects other workflows that communicate and cooperate with this workflow. In case of a *global* control flow failure and an induced abortion or suspension of a workflow, affected cooperating workflows are informed about this event. In contrast to this, a dynamic adaptation induced by a *local* control flow failure usually does not endanger a workflow's result as a whole but may violate temporal or quality constraints on which the cooperation partners agreed. Therefore, AGENTWORK derives the *temporal* implications of a dynamic adaptation by estimating the duration that will be needed to process the changed workflow definition (in comparison with the original definition). Furthermore, *qualitative* implications of the dynamic change are determined. For this purpose, so-called *quality measuring objects* are introduced.

This is described in Chapter 10 (*Handling Control Flow Failures for Cooperating Workflows*).

All mechanisms provided by AGENTWORK include that users may interact at any moment during the failure handling process. This includes the possibility of the user to classify as a control flow failure an event which has not been considered by the system. Furthermore, the user has the possibility to reject or modify workflow adaptations suggested by AGENTWORK.

5. ***A Prototypical Implementation Supporting the Integration of AGENTWORK into Distributed***

and Heterogeneous Environments (Contribution to requirement 4 in Section 1.3): Besides the prototypical implementation of the mechanisms above, this thesis describes an implementation supporting a CORBA³-based integration of AGENTWORK into the distributed and heterogeneous environments of real-world organizations such as hospitals or insurance business enterprises. Such an integration is viewed as essential in order to evaluate AGENTWORK under real-world conditions. The integration is mainly achieved by a generic mechanism that maps all AGENTWORK data objects to CORBA objects at execution time. Assuming that the databases and application programs of a real-world environment are also connected to CORBA, AGENTWORK can be integrated into such a new environment with a minimum of re-implementation. Only the “AGENTWORK Objects to CORBA Objects” mapping has to be adjusted w.r.t. the new environment.

As significant work has already been done w.r.t. relevant topics such as middleware-based architectures [BAKER 1997, SESSIONS 1998], event notification [COLLET ET AL. 1998, HANSON ET AL. 1998] and schema integration [LOPEZ ET AL. 1997, SANTUCCI 1998, MCBRIEN & POULOVASSILIS 1998, SCHMITT & TÜRKER 1998], the AGENTWORK CORBA approach of integrating heterogeneous and distributed data sources confines itself to combine existing approaches in an appropriate way.

The implementation of AGENTWORK and especially its integration into CORBA is described in Chapter 11 (*Implementation Issues*).

1.5 The HEMATOWORK System: Workflow Management in Hemato-Oncology

The primary application system to which AGENTWORK shall be applied is the system HEMATOWORK. This system is jointly developed by the Department of Computer Science and the Department of Medical Informatics, Statistics and Epidemiology of the University of Leipzig⁴ and addresses the workflow-oriented support of medical and administrative processes in the domain of hematooncology. This domain covers the diagnosis, therapy and follow-up of cancer diseases of the hematological and lymphatic system, such as leukemia (tumor of the white blood cells) and lymphoma (tumor of the lymph node system). In this section, we give a brief overview concerning HEMATOWORK.

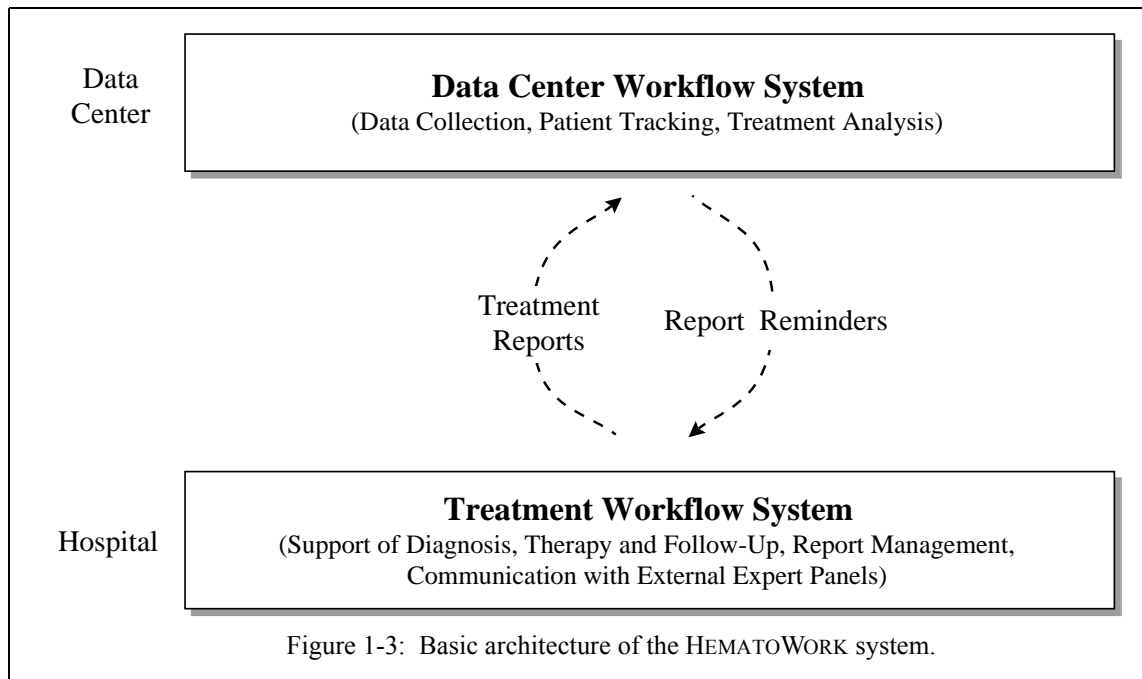
Basically, HEMATOWORK consists of two cooperating workflow systems (Figure 1-3). The first system is termed *treatment workflow system* and is designed to run at a hospital’s hematooncological ward and ambulance. The second system is called *data center workflow system* and is located at the data center collecting all patient data for statistical purposes.

The functionality of the **treatment workflow system** can be classified as follows:

- *Support of Cancer Diagnosis, Therapy and Follow-Up*: This mainly consists of the execution

3. CORBA = Common Object Request Broker Architecture.

4. HEMATOWORK is currently supported by the GERMAN RESEARCH ASSOCIATION (DFG) under grant number Ra 497/12-1.



of workflows based on the standardized treatment specifications for cancer. A typical workflow executed by this system and related to chemotherapy is shown in Figure 1-1.

- *Report Management*: This mainly consists of generating and editing treatment reports, in particular those that are needed by the data center to analyze the therapies of the different hospitals.
- *Communication with External Expert Panels*: This includes sending primary diagnoses or radiotherapy plans to expert panels for pathology respective radiology where these diagnoses and plans are reviewed, confirmed or revised.

The **data center workflow system** supports the data center w.r.t. the operational daily work of documenting and analyzing the therapies performed at the hospitals. Its functionality can be classified as follows:

- *Data Collection and Patient Tracking*: This covers the receipt, validation and storage of the reports sent by the treating hospitals. In particular, when a new report is received, the system checks whether all mandatory data fields have been filled out and whether the data is plausible. Furthermore, the system performs periodical checks whether all reports have been received for a particular patient. In case of missing, incomplete or implausible reports, the systems generates and sends reminders and admonitions to the responsible hospital.
- *Support of Treatment Analysis*: This mainly covers the conversion and movement of the report data to the programs and staff members performing the statistical analysis of the treatments.

One of the major open topics of the HEMATOWORK system is the concept and development of the *treatment workflow system*. To determine the requirements for this workflow system, the author of this thesis has interviewed physicians and has performed an analysis of frequently used hemato-oncological therapy specifications [HAVEMANN 1994, PFREUNDSCHUH & LÖFFLER 1994, DIEHL 1993, RIEHM 1995]. Furthermore, practical experiences of the author made when developing a therapy support system for pediatric oncology [MÜLLER ET AL. 1997 B] at the University Hospital of Mainz, Germany, have been used. The result has been that the treatment workflow system strongly should be able to cope with control flow failures as defined in Section 1.2, if the physicians shall accept it. For example, any of the used cytostatic drugs may have serious negative side-effects for several organs or tissues, so that workflow adaptations such as dropping or replacing the responsible drugs become necessary when these side-effects occur. In particular, these side-effects occur quite often so that workflow adaptations have to be performed for a significant group of patients which cannot be neglected. For a domain analysis motivating the need of dynamic workflow adaptation in cancer therapy we refer to [MÜLLER & HELLER 1998]. Similar results showing the need for adaptive workflow management in other medical domains, such as gynecology, have been found by [KUHN ET AL. 1995, REICHERT ET AL. 2000, DADAM ET AL. 2000]. For further details about the HEMATOWORK system and its preliminary results we refer to [MÜLLER & HELLER 1998, MÜLLER ET AL. 1998].

1.6 Structure of the Thesis

We complete this chapter with an overview of the thesis' structure. In Chapter 2, we discuss related work. This includes relevant work from the fields of commercial workflow products, advanced transaction management, dynamic and flexible workflow management, artificial intelligence planning systems, and agent technology. Then, before going into the details concerning the contributions listed in Section 1.4, Chapter 3 (*AgentWork Overview*) gives an overview concerning the principal AGENTWORK approach of handling control flow failures. In Chapter 4 (*Data and Rule Definition with ActiveTFL*), a motivation and overview concerning the logic-based data and rule definition language of AGENTWORK is given.

The specific research contributions listed in Section 1.4 are the topics of Chapter 5 to Chapter 11. This includes describing the workflow model in Chapter 5, workflow estimation in Chapter 6, the complete set of control actions in Chapter 7, structural workflow adaptation in Chapter 8 and Chapter 9, workflow cooperation in Chapter 10, and the prototypical implementation in Chapter 11.

The thesis concludes with a summary and outlook in Chapter 12.

This chapter discusses related work concerning the handling of workflow failures, in particular of *control flow* failures. It is organized as follows: In Section 2.1 we characterize current *commercial* workflow management systems and show that their failure handling capabilities are too restricted for our requirements. Section 2.2 discusses the contributions of several *advanced transaction models* for the handling of control flow failures. Section 2.3 sketches the possibilities and limitations of exception handling known from programming languages. In Section 2.4 we discuss relevant research approaches from the areas of *adaptive* and *cooperative* workflow management. Relevant work done in the field of artificial intelligence (i.e., planning, cooperative agents, and temporal reasoning) is discussed in Section 2.5. The chapter concludes with a summary in Section 2.6.

Related work which is very specific for some detailed aspects of this thesis will be discussed in the respective chapters later on. This includes approaches from active and deductive databases for event-condition-action rules and temporal algorithms for estimating workflow durations.

To better characterize related work approaches, Table 2-1 summarizes the requirements which have been identified in Chapter 1 as being essential for the handling of control flow failures. Requirement 4 (*Handling of Data and Event Distribution and Heterogeneity*) of Chapter 1 is not listed in this table as this thesis does not provide research contributions for this topic. Rather, to cope with events in distributed and heterogeneous environments we will use and combine existing approaches in a straightforward manner. Therefore, we discuss related work concerning this requirement in Chapter 11 (*Implementation Issues*).

Central Requirements		Subrequirements
1.	Representation of Failure Events and Control Actions	1.1 High Semantic Level of Event and Control Action Representation 1.2 Temporal Structure of Events 1.3 Temporal Structure of Control Actions 1.4 Integrity of Failure Rules 1.5 Authorization of Control Actions
2.	Translation of Control Actions into Workflow Execution Operations and Structural Adaptations	2.1 Workflow Abortion and Suspension 2.2 Support of Reactive Adaptation 2.3 Support of Predictive Adaptation 2.4 Consideration of Data Flow Implications 2.5 Consistency of Adapted Workflows 2.6 Efficiency of Adaptation
3.	Handling of Inter-Workflow Implications of Control Flow Failures	3.1 Determination of Temporal Implications 3.2 Determination of Qualitative Implications

Table 2-1: Requirements for the classification of related work.

2.1 Commercial Workflow Management Systems

On the market, several hundred commercial workflow management systems exist [KARK & KARL 2000]. However, it has commonly been identified that most commercial systems do not provide sufficient support for managing failure situations [CASATI & POZZI 1999, MOHAN 1996, KUHN ET AL. 1995]. For example, in case of *device*, *system*, and *transaction failures* many commercial systems simply perform a rollback on the database storing workflow control data such as the current values of activity parameters. As workflows typically represent long-term processes such a rollback is not acceptable in most contexts because too much work already done is lost.

Furthermore, “external” processes that have been triggered by the workflow system – such as application program invocations, application database updates, or message deliveries to workflow users – are usually not covered by the failure handling. Thus, in case of a failure the workflow system may be inconsistent as processes are still active that don’t make sense anymore at all or that at least operate on invalid data. One of the few exceptions in this context is IBM MQSERIES WORKFLOW [LEYMANN & ROLLER 2000] which runs on top of the message-oriented middleware IBM MQSERIES [IBM 2002 D]. This middleware provides application programming services that enable the workflow system and application programs to communicate asynchronously with each other

using messages. In particular, MQSERIES supports *transactional* messaging based on persistent message queues [BLAKELEY ET AL. 1995]. This means that operations on messages (e.g., message input, message processing, and message output) are processed as transactional units. In case of a workflow failure not only the affected workflow execution steps but also messages sent by the workflow system can be rolled back [LEYMANN 1997]. Broader efforts to handle the problem of such “external” processes by integrating commercial workflow systems into services or monitors for distributed ACID transaction processing have not yet been successful. For example, the OBJECT MANAGEMENT GROUP has specified a standardized CORBA workflow facility [SCHULZE ET AL. 1998, SCHULZE 1999] in order to integrate workflow systems into the 2-phase-commit protocol of the CORBA TRANSACTION SERVICE [OMG 2002 B]. However, this facility has not yet, to the author’s knowledge, resulted in any coupling of commercial workflow systems with the CORBA TRANSACTION SERVICE. Anyway, such approaches as provided by IBM MQSERIES WORKFLOW or the OBJECT MANAGEMENT GROUP do not solve the principal problem that a rollback – even if it covers “external” processes – often is not appropriate as too much work already done may be lost. Also the WORKFLOW MANAGEMENT COALITION has not suggested any advanced failure handling approach overcoming the limitations of current commercial systems.

Concerning *control flow* failures, the situation is similar. Only a few systems provide support: For example, PROMINAND [IABG 2002, KUBICEK & REICHERT 1996] views a workflow as a control flow specifying when a document folder has to be sent to which user (“electronic circulation folder”). This folder and the specifications which document-related activities have to be performed are modeled as *one* object migrating from one user to the next one [KARBE ET AL. 1990]. During this migration process, the system allows the user to skip, insert, or reorder activities dynamically. PROMINAND also provides a programming interface by which such a dynamic activity skip or insertion can be induced by external application programs. Thus, an integration of the system into larger contexts is facilitated. However, several limitations exist:

- First, the expressiveness of the workflow definition language is very poor. For example, loops are not supported. Furthermore, due to the document-oriented character the data flow is limited to the exchange of files between activities.
- Second, the set of adaptation operations is incomplete. In particular, no operation is provided to add new activities *parallel* to already existing activities so that activity sequences do not have to be split up. Furthermore, the supported adaptation operations consider only the control flow but ignore implications of an adaptation for the data flow.
- Third, only activities currently processing the circulation folder may be adapted. Thus, a predictive adaptation affecting activities not yet reached by the control flow is not possible.

The workflow management system INCONCERT [ABBOT & SARIN 1994, SARIN 1996, MOHAN 1996] allows the dynamic adding or removing of activities and of control flow dependencies as well. In addition to PROMINAND, INCONCERT includes event-condition-action (ECA) triggers in its workflow model. Triggering events may be certain workflow or activity state changes (e.g., an activity becomes ready for execution), external user-defined events, or temporal events such as an

elapsing deadline. Allowed actions may be user notifications, the start or abortion of workflows or activities, or application program calls. However, similar to PROMINAND, the expressiveness of the workflow definition language is very limited, and data flow implications of adaptations are not considered.

Other workflow management systems, such as COSA [SOFTWARE LEY GMBH 2000], ACTION REQUEST SYSTEM [REMEDY CORP. 2000], LOTUS DOMINO WORKFLOW [GIBLIN & LAM 2000], and PAVONE ESPRESSO [MOHAN 1996] offer a comparable functionality concerning workflow adaptation.

However, none of the mentioned commercial systems can be used to address the requirements listed in Table 2-1:

- First, although the mentioned systems provide some support concerning requirement 2.2 (*Support of Reactive Adaptation*) via ECA rules, they do not sufficiently address the *automation* of this process. Even if a trigger automatically derives that for example some activity has to be performed additionally, the user still has to decide *where* the new activity node shall be inserted into the control flow. For example, when several control flow paths are executed in parallel at the failure moment, the user has to select the path into which the new activity shall be inserted.
- Second, the requirements 2.3 (*Support of Predictive Adaptation*) and 3 (*Handling of Inter-Workflow Implications of Control Flow Failures*) are not supported by commercial workflow systems.
- Third, requirement 2.4 (*Consideration of Data Flow Implications*) is neglected seriously as necessary data flow adaptations are not derived when activities have been dropped or added.
- Fourth, although some commercial systems provide some functionality concerning dynamic adaptation, they do not allow external programs to *access* this functionality via a programming interface (except of PROMINAND). Thus, the coupling of such a commercial system with components for an advanced handling of control flow failures is very difficult.

A detailed discussion of commercial workflow systems and their limitations w.r.t. dynamic workflow adaptation can be found in [REICHERT 2000]. Due to these limitations, it has been decided *not* to build AGENTWORK on the basis of a commercial workflow management system.

2.2 Advanced Transaction Models

As already sketched in Chapter 1, the ACID transaction model [GRAY & REUTER 1993, HÄRDER & RAHM 2001] is not appropriate for workflow failure handling. This is mainly because workflows typically represent long-term processes so that too much work already done would be lost in case of a rollback. Furthermore, strict isolation of a database object for the whole life span of a workflow usually is not acceptable. Therefore, several advanced transaction models have been proposed relaxing the strict ACID properties in a workflow-oriented manner. Surveys on such advanced transaction models can be found, for example, in [ELMARGARMID 1992, JAJODIA & KERSCHBERG

1997, HÄRDER & RAHM 2001]. In our context, *partial backward recovery*, *compensation-based recovery* and *forward recovery* are of importance.

2.2.1 Partial Backward Recovery and Compensation-Based Recovery

To minimize the loss of work in case of a failure, *partial backward recovery* relaxes atomicity and rolls back a workflow not to its starting point but only partially (*partial undo*). For example, when a failure occurs during a booking workflow only the activities that dealt with the hotel booking may be rolled back while prior work done for the flight booking may not be affected. When the failure has been resolved the workflow is restarted at that point to which it has been rolled back. Typically, so-called *savepoints* are specified at workflow definition time to indicate to which point a workflow shall be rolled back in case of a failure [GRAY & REUTER 1993, GREFEN ET AL. 1999 B].

Orthogonal to this, *compensation-based backward recovery* has to be performed in case that isolation is relaxed and changes performed by a workflow are released already during its execution. Thus, a rollback cannot be performed anymore by a state-oriented undo recovery, but only by a logical undo through *compensation*. Such a compensation-based backward recovery which goes back to the SAGAS model [GARCIA-MOLINA & SALEM 1987, GARCIA-MOLINA ET AL. 1991] can be characterized as follows: First, typically each workflow activity is viewed as an ACID transaction. Second, to every activity a *compensation activity* is assigned which is able to logically undo or at least to minimize the effects caused by the respective workflow activity. For example, for a hotel booking activity a compensation activity could *cancel* the hotel booking by generating and sending a cancellation fax to the hotel. Third, in case of a failure the currently executed activities are rolled back by a conventional undo recovery, and for every already committed activity the respective compensation activity is performed. This is usually done in the reverse order of the already executed control flow.

Based on these two general recovery models, several workflow failure handling approaches have been suggested. For example, Leymann proposes a partial backward recovery model based on the concept of so-called *compensation spheres* [LEYMANN 1995, LEYMANN & ROLLER 2000]. A compensation sphere is a set of workflow activities which has to be either executed successfully as a whole or – in case of a failure – has to be rolled back. This sphere roll back is performed by executing compensation activities for all sphere activities that have already been executed. A workflow may be hierarchically divided into multiple compensation spheres to achieve an application-oriented partitioning for failure handling purposes. Dependencies between spheres in one workflow or even in different workflows are also considered. For example, a sphere S_1 in a workflow W_1 may have produced data that have been already been consumed by a sphere S_2 in a workflow W_2 . Then, if S_1 has to be compensated because of a failure, S_2 is compensated as well (*cascading compensation*) since it operated on data not being valid anymore because of the failure of S_1 . A limitation of the approach is that it does not consider inter-workflow dependencies as spheres depending on each other have to belong to the same workflow. Furthermore, to the author's knowledge no implementation of compensation spheres has been provided so far.

Several variations and extensions of partial backward recovery and compensation-based recovery

have also been suggested, such as the dynamic generation of compensation plans [DAVIS ET AL. 1996, ALONSO ET AL. 1996] or optimization issues to avoid unnecessary compensation steps [KAMATH & RAMAMRITHAM 1998].

2.2.2 Forward Recovery

To reduce the loss of work due to failures as much as possible, so-called *forward recovery* approaches attempt to avoid a backward recovery whenever this is possible [DAYAL ET AL. 1991, WÄCHTER & REUTER 1992, WORAH & SHETH 1997]. For example, in the CONTRACTS model [WÄCHTER & REUTER 1992, REUTER & SCHWENKREIS 1995, REUTER & SCHWENKREIS 1995] it can be specified that in case of an activity failure this activity shall be repeated or that an alternative activity shall be executed. This allows to react on failure situations in a more application-specific manner than as with a backward recovery. The latter is only performed in case that the specified failure handling fails as well (e.g., if the repeating of the activity or the execution of an alternative activity fails).

To handle system failures, the CONTRACTS system first logs the entire workflow execution history such as the chronology of user inputs and of the activity input and output data persistently in a transactional database (*context management*). Second, by using recoverable message queues [BERNSTEIN ET AL. 1990] it is also achieved that in failure cases the loss of information exchanged with workflow clients is reduced to a minimum. When a failure such as a server crash occurs, the CONTRACTS system interrupts the workflow and – after the failure has been resolved – continues it by restoring the last consistent execution state logged by the context management. The workflow then is continued according to the specified workflow definition.

We omit other details of CONTRACTS such as isolation predicates [WÄCHTER & REUTER 1992], as they are not relevant for the handling of control flow failures.

Another forward-oriented recovery approach has been realized within the TAM¹ system [LIU & PU 1998 A, ZHOU ET AL. 1999]. TAM provides several constructs to specify interaction dependencies between activities in an application-dependent manner, such as an '*A precede B*' dependency between two activities *A* and *B*. These dependencies can dynamically be restructured if exceptions occur. For example, for a software engineering process there may be two main activities, namely the specification activity *S* (consisting of two subactivities *S₁* and *S₂*) and the implementation activity *I* (consisting of three subactivities *I₁*, *I₂*, and *I₃*). At definition time, the only interaction dependency specified is that *S* should precede *I*. Suppose now *S₁* has been completed, but *S₂* has to be postponed for days or weeks because some important information required from a remote site is not available. This may cause the problem that subactivities of *I* which depend only on *S₁* (e.g., *I₁* and *I₃*) have to wait for the completion of the entire activity *S* (in particular for *S₂* in this case), even though they do not depend on *S₂*. To cope with this, TAM allows to restructure the '*S precede I*' dependency by breaking it down to a number of subactivity dependencies such as '*S₁ precede I₁*',

1. TAM = Transactional Activity composition Model

' $\{S_1, S_2\}$ precede I_2 ', and ' S_1 precede I_3 '. Then, I_1 and I_3 can be started when S_1 has finished as they do not depend on S_2 . Furthermore, any activity may be dynamically split up (e.g., S_2 may be split up to S_{21} and S_{22}), and for the subactivities additional dependencies may be defined.

2.2.3 Discussion

At first glance, the advanced transaction approaches described above can be used to handle *control flow failures*, though they have not primarily been proposed for this failure type. For example, the four chemotherapy drug administrations of the workflow in Figure 1-1 (CYCLOPHOSPHAMID, DOXORUBICIN, VINCRISTIN, PREDNISOLON) could be viewed as a compensation sphere according to [LEYMANN 1995]. Compensating activities such as giving an antibiotic drug or making an additional diagnostic examination could be assigned to activities of this chemotherapy sphere. When a failure event such as a critical hematological toxicity occurs, the sphere could be rolled back to its beginning by executing the above-mentioned antibiotic or diagnostic activities.² Alternatively, in terms of the CONTRACTS model one could specify an antibiotic drug activity or an diagnostic examination activity as *alternative* activities that should be performed in case that one of the chemotherapy administrations fails. Concerning TAM, the dynamic splitting of activities could be used to add activities to a process at execution time. Thus, at least some primitive *reactive* "adaptation" of the control flow can be achieved (requirement 2.2 in Table 2-1).

However, handling control flow failures on the basis of these advanced transaction models is not appropriate because of the following reasons:

- First, the type of an appropriate compensating or alternative activity very often will depend on the type of the failure event. For example, the administration of an antibiotic drug may be an appropriate compensating or alternative activity in case of a *hematological* toxicity as it helps to prevent a severe infection due to hematological toxicity. But it usually is useless or even dangerous concerning other event types such as a *renal* toxicity, as then every additional drug would only endanger the renal system. Thus, *conditional* compensating or alternative activities would have to be specified which is not yet supported sufficiently by current approaches.
- Second, even if we allow conditional compensating, alternative activities, or dynamical restructuring, requirement 2.3 (*Support of Predictive Adaptation*) from Table 2-1 is not met. This is because none of the above-mentioned transaction approaches copes with any part of the control flow that has not yet been scheduled for execution at the moment of the failure. Therefore, a control action such as *drop* or *replace activity of type A for the next two weeks* cannot be handled predictively.
- Third, not all types of control actions that may become necessary can be realized by these advanced transaction approaches. For example, depending on the transaction approach it is difficult to achieve that in case of a failure a single medical activity is *dropped* or *delayed* while

2. The antibiotic activity is suitable to compensate a hematological toxicity as it protects from infections fostered by this type of toxicity. The diagnostic activities are suitable as they provide more information about the toxicity.

the rest of the workflow is continued as originally specified.

- Fourth, approaches such as TAM do not sufficiently support the automation of failure handling. In particular, the restructuring of the activities has to be done manually. This may be suitable for software engineering applications, where only a small number of high-level activities is modeled. However, it is not appropriate for applications such as cancer therapy where processes are modeled on a very detailed level and consist of many activities.

Overall, it can be stated that advanced transaction models do not provide sufficient support for handling control flow failures, although they consider much more application semantics than the traditional ACID model. However, they form a basis for the control flow failure handling approach suggested in this thesis. For example, a compensation-based backward recovery may become necessary in case of a *global* control flow failure when the continuation of a workflow does not make sense anymore *at all* and the workflow therefore has to be aborted. Furthermore, workflow suspension as supported by CONTRACTS may be useful in case a workflow has to be suspended due to some control flow failure. As the mechanisms described, for example, by [ALONSO ET AL. 1996, KIEPUSZEWSKI ET AL. 1998, GREFEN ET AL. 1999 B, WÄCHTER & REUTER 1992] are sufficient for this purposes, we do not focus on the transactional aspects of workflow abortion and suspension in this thesis.

Table 2-2 summarizes our discussion of advanced transaction models.

2.3 Exception Handling in Programming Languages

A workflow definition can be viewed as some sort of a high-level program where the activities form the different program “operations“, and where edges and control flow nodes specify the program’s control and data flow. Thus, it makes sense to inspect whether *programming languages* contribute to the problem of handling control flow failures. In this context we concentrate on *procedural* languages (e.g., C++, JAVA) as these languages are characterized by an explicit control flow notion. Among these procedural languages, we discuss the exception handling approach of JAVA [WOLFF 1999], as it can be viewed as one of the most advanced ones, and as exception handling has been integrated into the language from the beginning (in contrast to, for example, C++). We do not discuss the C++ handling approach, as it is very close to that of JAVA. Other programming language types such as functional languages (e.g., LISP) or logic languages (e.g., PROLOG) are not discussed, as with them a program is not primarily specified by stating the control flow but by giving a number of functions (LISP) respective rules and predicates (PROLOG).

In JAVA, exception handling is specified by so-called *try* and *catch* blocks:

- A *try* block “tries“ to perform some code. In case something fails, it generates an exception object which contains information about the failure type and which is “thrown“ (i.e., sent) to some *catch* block to handle the failure. e.g.,

Requirements for Handling of Control Flow Failures	Supported	Representative Approaches and Remarks
1.1 High Semantic Level of Event and Control Action Representation 1.2 Temporal Structure of Events 1.3 Temporal Structure of Control Actions 1.4 Integrity of Failure Rules 1.5 Authorization of Control Actions	No (out of scope)	
2.1 Workflow Abortion and Suspension	Yes	Backward recovery (for abortion) CONTRACTS (for suspension)
2.2 Support of Reactive Adaptation	Limited	By conditional compensating activities (for compensation-based recovery), by conditional alternative activities (for forward recovery), or by dynamic activity restructuring (TAM)
2.3 Support of Predictive Adaptation 2.4 Consideration of Data Flow Implications 2.5 Consistency of Adapted Workflows 2.6 Efficiency of Adaptation	No (out of scope)	
3.1 Determination of Temporal Implications 3.2 Determination of Qualitative Implications		

Table 2-2: Support of advanced transaction models.

try { some statements “trying“ to generate an object *t*

```

    if (t == null)                                // if failure,
    throw new NullPointerException(some detail parameters); // generate and sent
    }                                                // an exception object

```

To define exception objects, JAVA provides predefined base classes (from which, for instance, the above class *NullPointerException* can be derived).

- A *catch* block has to be placed directly after a *try* block and specifies how to handle an exception of a given class, e.g.,

```

catch(NullPointerException e) { // type declaration of catch block

// handle the null pointer exception using information encoded in e }

```

The assignment of *catch* blocks to *throw* statements is done type-based, i.e., the first *catch* block having a type declaration compatible with the thrown exception is selected.

The main advantage of this approach is that it allows to structure exception handling in a readable manner as the code identifying the exception is clearly separated from the code where the exception is handled. Furthermore, the existence of predefined base classes for exception objects facilitates exception handling programming.

However, this exception handling approach is not appropriate for handling control flow failures. The main argument is the same made w.r.t. the suggestion in Section 1.2.3 why not to cope with control flow failures by adding *conditional branches* to a workflow definition: Exceptional events causing control flow failures typically may occur *asynchronously at any time* during workflow execution. However, handling them by a “*try & catch*” approach would require that the relative point in time of their occurrence w.r.t. a particular position in the program (i.e., the workflow definition) is known at workflow definition time. However, this is not possible for most types of failure events.

Note that this does not mean that the JAVA exception handling approach cannot be used to *implement* a failure handling approach for workflow systems or to handle exceptions in a more restricted context, such as system exceptions during the execution of single activities. It only means that the JAVA exception handling approach is not appropriate on the conceptual level to handle control flow failures as defined in Chapter 1.

2.4 Workflow Research Approaches

In this section, we discuss workflow research approaches, namely approaches from the fields of adaptive workflow management (2.4.1) and cooperative workflow management (2.4.2).

2.4.1 Adaptive Workflow Management

Research has widely identified that workflow management systems should be able to react on changing workflow environments and application situations in a more flexible manner than it is currently possible with commercial workflow products and advanced transaction models [SHETH 1997, REICHERT & DADAM 1997]. Consequently, during the last years several workflow adaptation models and prototypes have been proposed to overcome these limitations [REICHERT & DADAM 1998, ELLIS ET AL. 1998, WESKE 1999 B, CASATI ET AL. 1999]. Existing approaches can be classified into three main categories [HEINL ET AL. 1999, DADAM ET AL. 2000]:

1. *Late Modeling/Late Binding Adaptation*: To achieve more flexibility, approaches of this category omit or “underspecify” several aspects of a workflow definition, such as the particular sub-workflow that shall be invoked at a certain control flow position, or the particular order in which an activity sequence shall be executed [JABLONSKI ET AL. 1997, HAGEMEYER ET AL. 1997, MÜLLER & HELLER 1998]. Some approaches also leave open which particular *resources* (e.g., programs, users, devices) shall be allocated for an activity execution [HAN ET AL. 1996, LIU & PU 1997, CHIU ET AL. 1999]. The aspects left open are determined at execution time, e.g., when

data has become available that is needed to decide which particular subworkflow shall be used. Syntactically, open aspects are often represented by some abstract placeholder nodes or edges in the workflow definition. The overall limitation of the approaches of this category is that it has to be known at workflow definition time *what* has to be left open, and *when* the open aspect has to be decided, i.e., the relative workflow position of the placeholder representing the open aspect has to be known beforehand. Furthermore, depending on the degree of late modeling, the user has to be familiar with the workflow definition language.

2. *Ad Hoc Adaptation (Instance Adaptation)*: Approaches of this category assume that it may become necessary at any moment during workflow execution to adapt the workflow in an ad hoc way that cannot be pre-modeled at workflow definition time and therefore cannot be covered by late modeling. Such approaches usually provide a set of generic graph operators supporting as many types of structural adaptations as possible [REICHERT & DADAM 1998, CASATI ET AL. 1999]. The control flow failure handling approach of AGENTWORK belongs to this category. The principal challenge of ad hoc adaptation approaches is first to guarantee that an adaptation leads to a *correct* workflow, and second the question to what degree such an adaptation can be automated.
3. *Workflow Definition Adaptation ("Schema Evolution")*: Approaches of this category address the adaptation not of a single running workflow but of a workflow *definition* (i.e., a workflow schema or schema) so that all workflows based on this definition may be affected [REICHERT ET AL. 1998, JOERIS & HERZOG 1998, KRADOLFER & GEPPERT 1999]. This becomes necessary if it is detected that a workflow definition *generally* is inadequate, independently from the particular context in which it is used. For example, in a medical context it may be detected that a certain drug must not be given anymore to *any* patient as dangerous side-effects may have become known so that the drug is taken from the market. Thus, *all* workflow definitions consisting of activities administering this drug have to be changed in the workflow definition database (e.g., by removing the respective activity nodes). An important subproblem of workflow definition adaptation is how to cope with workflow instances that have been started on the basis of an *old* workflow definition which is no longer valid anymore as it has been changed in the workflow definition base.

We now discuss relevant approaches of these categories. Workflow failure handling and adaptation approaches that originate from the fields of artificial intelligence planning and agent technology [DELLEN ET AL. 1997, BECKSTEIN & KLAUSNER 1999A, KLEIN & DELLAROCAS 2000 B] are discussed in Section 2.5 (*Artificial Intelligence Approaches*).

MOBILE: MOBILE is a modular workflow management system mainly addressing late modeling adaptation. It consists of several modules for orthogonal workflow *perspectives*, such as the so-called *behavior* perspective (i.e., when what is executed in the control flow) or the *information* perspective (i.e., what data is consumed and produced) [JABLONSKI ET AL. 1997]. In MOMO, the so-called MOBILE MODELING language, workflow definitions can be left *incomplete* if, for

example, the suitable activity, activity order or subworkflow can only be determined at execution time [HORN & JABLONSKI 1998, HEINL ET AL. 1999]. In particular, *goal descriptions* can be assigned to a workflow definition restricting the types of activities, activity orders or subworkflows that may be selected at execution time (i.e., only such selections are finally allowed which do not violate the workflow goal). This late modeling approach is termed as *descriptive modeling* [JABLONSKI ET AL. 1997], in contrast to *prescriptive modeling* which would have to specify all aspects entirely at workflow definition time. Descriptive modeling enables the MOBILE system to better cope with situations that cannot entirely be foreseen at workflow definition time. However, in MOBILE the *user* has to decide how a workflow shall be completed concerning aspects left open at workflow definition time. Methods to *automatically* derive the workflow completion when, for example, relevant data have become available are not discussed by the authors. In particular, *correctness criteria* constraining possible completions to avoid inconsistent workflows are not provided. Furthermore, as with all late modeling approaches, *asynchronous* failure events occurring during workflow execution cannot be handled appropriately. This is because their relative point in time of occurrence w.r.t. the workflow control flow usually is not known in advance.

ADEPT_{FLEX}: In the context of the ADEPT³ workflow project, the ADEPT_{FLEX} model has been proposed to support ad hoc workflow adaptation [REICHERT & DADAM 1998, DADAM ET AL. 2000]. In particular, ADEPT_{FLEX} contributes to the following topics:

- First, it provides a *block-oriented* workflow definition language. This means that activity sequences, parallel and conditional branching, and loops are specified as symmetrical blocks with well-defined start and end nodes. These blocks may be nested, but they are not allowed to overlap. As an extension, synchronization edges as well as failure backward edges can be defined between activity nodes. Furthermore, one can specify different modes of parallel execution, such as that the parallel execution terminates either when the *first* path commits, or only when *all* execution paths commit. Additionally, the data flow between activities can be specified explicitly.

On one side, this block-oriented model allows to define well-structured and readable workflows. On the other side, the expressiveness needed for a broad range of applications is achieved. Furthermore, a workflow definition is verified on the basis of *correctness* and *consistency constraints*. For example, it is verified whether deadlocks may occur or whether an activity node will be provided with all needed input objects.

- Second, ADEPT_{FLEX} provides a complete operator set for the dynamic insertion, dropping or moving of activities or activity blocks at any point in time during workflow execution. In particular, workflow regions which have not yet been entered by the control flow can be changed as well (in contrast to, for example, PROMINAND). Furthermore, the execution of dynamically inserted activities can be synchronized with the execution of already existing ones. In particu-

3. ADEPT = APPPLICATION DEVELOPMENT BASED ON ENCAPSULATED PREMODELED PROCESS TEMPLATES

lar, ADEPT_{FLEX} offers operators to adjust the data flow in case of a dynamic adaptation. For example, such an operator may link a dynamically inserted activity to existing workflow data elements. Finally, operator subsets are aggregated to high-level “semantic” operators which allow the user to perform adaptations without having to take care about details on the node and edge level.

- Third, ADEPT_{FLEX} distinguishes between *permanent* changes which are preserved until the workflow completion, and *temporary* changes which are retracted once the changed region has been executed. This is of importance for long-running workflows, where changes may affect regions that are entered several times, e.g., due to a partial workflow rollback or due to loop iterations supporting cyclic processes such as the one in the chemotherapy example in Figure 1-1.
- Fourth, based on the correctness and consistency constraints mentioned above, it is checked efficiently whether an adaptation may lead to an incorrect workflow (e.g., to a workflow with a deadlock or an activity node without the needed input objects). If this is the case, the systems suggests “repair” operations such as inserting additional data flow edges for missing input objects.
- Fifth, ADEPT_{FLEX} also addresses the dynamic adaptation of *distributed* workflows which may be controlled by multiple servers [BAUER ET AL. 2001].

Furthermore, the ADEPT_{TIME} component [DADAM ET AL. 2000] supports the management of temporal workflow constraints and determines the effects a dynamic adaptation might have for such constraints. At workflow definition time a minimal and a maximal duration can be specified for every activity. Temporal constraints can be assigned to activity nodes specifying that for example a node has to be reached not later than 48 days after workflow start. The duration information is then used first to verify at definition time whether a workflow is temporally consistent (e.g., whether an activity deadline is realistic at all w.r.t the durations of the predecessor activities). Second, it can also be estimated at execution time whether a dynamic adaptation would violate any temporal constraint assigned to an activity. If this is the case, the system informs the user about the conflict and offers different strategies to resolve it, such as aborting the adaptation or relaxing some temporal constraints. The temporal estimations are based on the Floyd-Warshall algorithm [DECHTER ET AL. 1991].

Additionally, in [HEINLEIN 2001, HEINLEIN 2000] so-called *interaction expressions* are used to specify inter-workflow dependencies in the ADEPT context. For example, such an interaction expression could specify that a patient cannot be present in two examination rooms at the same time, so that two activities of two different diagnostic workflows for this patient may have to be serialized. Generally, these interaction expressions can also be used to add “logical” constraints to the “syntactical” correctness constraints of ADEPT_{FLEX}. Such a logical constraint could then forbid the dynamic insertion of a drug activity *A* if some incompatible drug activity *B* already shall be performed in the neighborhood.

The overall contribution of the ADEPT_{FLEX} approach is its comprehensive, detailed and formal operator framework for dynamic workflow adaptations and its focus on correctness and consistency

issues. However, ADEPT_{FLEX} does not provide algorithms that *automatically* decide under which circumstances which structural adaptations should be applied to a workflow instance. Concerning the determination of *temporal* implications of a dynamic adaptation, it can be argued that for more precise temporal estimations not only durations specified at definition time but also activity duration measurements of prior workflow executions should be used. Furthermore, it is not discussed how a dynamic adaptation may violate inter-workflow constraints specified via the interaction expressions described above.

To summarize in terms of Table 2-1, ADEPT_{FLEX} provides a clear and detailed formal foundation concerning requirements 2.2 (*Support of Reactive Adaptation*), 2.4 (*Consideration of Data Flow Implications*) and 2.5 (*Consistency of Adapted Workflows*) and also a valuable approach for requirement 3.1 (*Determination of Temporal Implications*). However, ADEPT_{FLEX} does not address the *automation* of dynamic adaptations and does not meet requirements 2.3 (*Support of Predictive Adaptation*) and 3.2 (*Determination of Qualitative Implications*).

CHIMERA-EXC: To cope with so-called *expected exceptions* in the WIDE workflow project [GREFEN ET AL. 1999 A], Casati et al. have developed the rule-based exception handling language CHIMERA-EXC [CASATI 1998, CASATI ET AL. 1999]. CHIMERA-EXC is an object-oriented extension of the logic-based programming language DATALOG [CERI ET AL. 1989] and has been integrated into the FAR⁴ workflow system of WIDE. FAR itself is based on the commercial workflow management system FORO [SEMA GROUP 2000].

In the terminology of the authors, an expected exception is an event which is known to occur sometimes asynchronously during workflow execution and therefore requires some ad hoc handling, such as sending an additional message to a user or performing an additional activity. Therefore, expected exceptions correspond to our notion of control flow failure events. To cope with them, CHIMERA-EXC supports ECA rules to monitor events and to derive appropriate actions. The following event types are supported: *Data manipulation events* deal with operations changing the contents of a database (such as a table insert or update). *External events* are raised by external applications. *Workflow events* cover, for example, the starting, completion or abortion of workflows or workflow activities. *Temporal events* are defined as patterns on a time axis. For example, a temporal event could be that the calendar date reaches the 20th September 2000 or that 60 days have been elapsed since the 1st January 2000. The *condition* part of a CHIMERA-EXC rule states when an event has to be viewed as an exceptional event requiring further handling.

The *action* part of a CHIMERA-EXC rule includes data manipulation actions and operations on the workflow and activity level. The latter category includes primitives to start, rollback or suspend workflows, or to reject, cancel or add activities. For example, in case of a deadline expiration in a billing workflow such an action part could add an activity to this workflow sending a bill reminder to the customer who has not yet paid the bill. Additionally, in [CASATI ET AL. 1998] constraints are introduced specifying when an adapted workflow is viewed as correct. Such a constraint for exam-

4. FAR = Foro Active Rules

ple specifies that only activities after a conditional branching may have entry conditions as otherwise – because of the missing alternatives – a violated entry condition may lead to a deadlock.

In [CASATI ET AL. 1996, CASATI 1998] the proposed approach is also used for an event-based protocol for logical workflow interoperability and cooperation in inter-organizational contexts. Inter-workflow cooperation is specified at the communication level. By so-called SEND and RECEIVE nodes, workflows can synchronously exchange information about results via an event notification service. Failures leading to a workflow *starvation* (i.e., a receiver workflow waits in vain for a result) or a *deadlock* (i.e., two workflows in vain wait for results from each other) can be handled as follows: When a waiting threshold expires, either an alternative control flow path that already has been specified at definition time is executed, or the conflict is resolved manually in an ad hoc manner.

Overall, CHIMERA-EXC provides a comprehensive active rule approach to automate the handling of logical failures and control flow failures in particular. However, several limitations exist:

- First, the temporal dimension of triggered actions is not supported sufficiently. For example, when an action states that an activity has to be performed additionally, it is not specified whether this shall be done immediately after the rule has been triggered or somewhere later on. An immediate execution sometimes may not be possible because of missing data or unavailable resources. In particular, control actions such as dropping a particular activity *for the next 7 days* or *until some termination condition holds* are not supported.
- Second, concerning the *translation* of control actions into structural workflow adaptations, the approach only supports a *reactive* strategy. *Predictive* adaptations based on temporal estimations are not addressed.
- Third, the correctness criteria stating when an adapted workflow is viewed as correct are purely syntactical and do not cover logical restrictions such as that two certain medical activities must not be performed in parallel.
- Fourth, the way the system adjusts the *data flow* after a control flow adaptation remains unclear. Though in [CASATI ET AL. 1998] some low-level data flow constraints such as type compatibility conditions are introduced to support data flow consistency, no algorithm is described that computes necessary data flow adaptation in case of a dynamically adapted control flow.
- Fifth, rule integrity is not explicitly addressed. Though CHIMERA-EXC is based on DATALOG and therefore has a logical foundation, it is not said how the underlying logic calculus can be used to achieve rule integrity for the specific failure handling context. This is a critical point, as already for simple processes large rule sets may be necessary for exception handling. As a consequence, at execution time undesirable side-effects such as contradicting conclusions may occur.
- Sixth, concerning inter-workflow cooperation the proposed failure-handling strategy only covers that a result is not provided *at all* by a cooperating partner. A more detailed communication protocol informing that because of a control flow failure a result will be provided *later* or *with reduced quality* is not supported. In particular, the authors do not investigate mechanisms

adapting a receiver workflow itself dynamically so that it can better cope, for example, with a result of reduced quality.

Summarizing in terms of Table 2-1, CHIMERA-EXC provides useful contributions w.r.t. requirements 1.2 (*Temporal Structure of Events*), 2.2 (*Support of Reactive Adaptation*) and 2.5 (*Consistency of Adapted Workflows*). However, CHIMERA-EXC does not sufficiently address requirements 2.3 (*Support of Predictive Adaptation*), 2.4 (*Consideration of Data Flow Implications*) and 3 (*Handling of Inter-Workflow Implications of Control Flow Failures*).

MOKASSIN: In contrast to approaches such as ADEPT_{FLEX} [REICHERT & DADAM 1998], MOKASSIN does not focus on the question which operator set should be provided for workflow adaptation and which formal correctness criteria should be considered. Rather, the focus is to keep the execution behavior of a workflows as flexible as possible and as much as configurable by providing an *extensible* workflow meta-model [JOERIS 1999, JOERIS & HERZOG 1999, GRONEMANN ET AL. 1999]. Additionally, the applicability of an adaptation is not coupled with global correctness criteria that have to be met, but depends on the particular workflow and execution context.

The most important modeling primitive in MOKASSIN is the *task*. Its definition consists of an interface as well as a number of implementations from which one is dynamically selected at execution time. The task interface specifies the attributes and parameters of the task and describes its context-free behavior. A task implementation is either a user-defined program or a process model. The latter is graphically specified by a task network where the nodes represent the tasks and the edges the control and data flow dependencies between them. The modeler can influence the execution behavior by adapting the context-free behavior of single tasks or by extending the workflow meta-model by user-defined constructs in a context-sensitive manner. The context-free behavior of a task is specified by a number of possible execution states and – on the basis of ECA-rules – the valid transitions between them. The context-sensitive behavior of a task results from its control flow dependencies to other task nodes in the network. These dependencies can be adapted by assigning user-defined rules to a task which are then interpreted at execution time by the workflow management system. While these mechanisms of adapting the context-free and context-sensitive behavior of tasks is done at workflow definition time, dynamic ad hoc adaptation of a single workflow instance is possible as well. This can be done by a graphical editor, which offers a number of simple adaptation operations, e.g., to add or drop single nodes.

Overall, MOKASSIN provides a strong rule-based mechanism to extend the workflow meta-model by user-defined constructs and thus to increase flexibility. However, a serious disadvantage is that the authors do not provide any approach to analyze the user-defined extensions of the meta-model in terms of correctness. In particular, the integrity of user-defined rule sets is not checked. Furthermore, if compared with other approaches such as ADEPT_{FLEX}, the operator set for ad hoc adaptations is very restricted.

FLOW NETS, WORKFLOW NETS: In [ELLIS ET AL. 1995, ELLIS ET AL. 1998], the authors describe an approach for structural workflow adaptation based on petri nets [PETRI 1962]. For workflow

modeling and control, a special subtype of petri nets, so-called *flow nets*, is used. At execution time, each flow net can control several workflow instances. With a graphical editor, structural adaptations of a flow net can be performed, such as that two activities which have been executed sequentially so far now have to be executed in parallel [ELLIS & MALTZAHN 1997]. The correctness of the adapted networks is guaranteed in terms of reachability analysis of the petri net theory [BAUMGARTEN 1996]. However, the authors do not provide a set of predefined adaptation operators or a high-level adaptation interfaces, so that the person performing the adaptations has to know the syntactical details. Furthermore, as a flow net usually controls several workflow instances, the direct adaptation of single workflow instances is not possible. In addition to this, the authors do not consider the adjustment of data flows that may become necessary when a flow net is adapted.

[AALST 1997, VOORHOEVE & AALST 1997] describe an approach where – in contrast to [ELLIS ET AL. 1995] – every workflow instance is controlled by exactly one so-called *workflow net*, which is a workflow-oriented petri net subtype. Thus, ad hoc adaptation of single instances becomes possible. For this, a number of predefined transformation rules is provided, e.g., to refine an activity with a subworkflow, or to split up sequences to several paths executed in parallel or conditional, and to join them again. However, the limitation of this approach is that data flow aspects are neglected, and that the handling of loops remains unclear.

Furthermore, structural ad hoc adaptations of petri nets usually require the reconfiguration of the net *state* (or marking), i.e., of the token distribution [BAUMGARTEN 1996] within the net. However, many petri net-based adaptation approaches (e.g., [ELLIS ET AL. 1995, VOORHOEVE & AALST 1997]) assume that users are familiar with the petri net modeling language or with a graphical petri net tool. However, one cannot expect that for example physicians or nurses during clinical routine are able to change a token distribution of a petri net manually and to overlook all implications.

Several other approaches addressing adaptive workflow management exist, too. However, as they do not provide additional aspects relevant for this thesis, we do not discuss them here but refer to the respective literature [EDMOND & HOFSTEDE 2000, KRADOLFER & GEPPERT 1999, BORGIDA & MURATA 1999, KOKSAL ET AL. 1999, WESKE 1999 B, LIU ET AL. 1998 A].

Summing up, current approaches from the field of adaptive workflow management do not yet address the *automation* of control flow failure handling sufficiently, as they usually require a substantial amount of user input. Furthermore, the support of requirements 1.3 (*Temporal Structure of Control Actions*), 2.3 (*Support of Predictive Adaptation*), 3.1 (*Determination of Temporal Implications*) – except ADEPT_{TIME} – and 3.2 (*Determination of Qualitative Implications*) is very limited.

2.4.2 Cooperative Workflow Management

We now inspect how approaches from the field of *cooperative* workflow management support requirement 3 (*Handling of Inter-Workflow Implications of Control Flow Failures*). Generally, cooperation and collaboration aspects of workflow systems have already been identified as essential for workflow technology [DOGAC ET AL. 1998, GEORGAKOPOULOS ET AL. 1999 A], especially for process-oriented e-commerce applications and virtual enterprises [GREFEN ET AL. 2000, PAPA-

ZOGLOU & TSALGATIDOU 1999, RIDE 1999]. Research and vendor efforts have focussed on topics such as *interoperability frameworks* [FISCHER 2002, BUSSLER 1998], *workflow interaction modeling* [HEINLEIN 2001, DAVULCU ET AL. 1999, NGU 1999], *workflow synchronization* [ATTIE ET AL. 1996, TANG & VEIJALAINEN 1995], *collaboration management infrastructures* [BAKER ET AL. 1999], and workflow-oriented *event notification systems* [GEPPERT ET AL. 1998]. However, only a few approaches deal with failure management in inter-workflow cooperation scenarios. In particular, the implications of *control flow* failures for workflow cooperation scenarios have not yet been addressed sufficiently.

[HAGEN & ALONSO 1999] describe an approach for event-based communication between processes cooperating in a consumer-producer relationship. In the terminology of the authors, *events* are typed and parameterized signals raised by a running process to inform other processes about certain situations occurring during its execution (such as *product available* or *product not available* in an order handling scenario). During process execution a *dependency graph* is maintained storing which processes have been triggered by which events. The failure handling approach is based on the OPERA⁵ system [HAGEN & ALONSO 1998] and has been applied, for example, to the workflow-based e-commerce system WISE [LAZCANO ET AL. 2000]. It works as follows: Exceptions are viewed as special events which are generated in case of a process failure. Exception events are typed and have a parameter list so that they can carry information about the failure type and its circumstances. In case that a process P fails, the system derives from the dependency graph which processes depend on P and sends exception events informing them about the occurred failure type. The notified processes then may perform an abort or compensation-based partial rollback.

Generally, the proposed approach provides a useful event-oriented failure handling model for inter-dependent processes and is suitable for workflow cooperation scenarios such as distributed order processing in retail companies. However, it is not suitable for handling failure events in more complex scenarios such as cooperative medical care because of the following reasons: First, disseminating an exception event E that occurred to a “producer” process P_1 to a “consumer” process P_2 may be meaningless for P_2 as P_2 may not be able to derive the implications of E for itself. Rather, P_2 often is more interested in obtaining some “predictive” failure information about the *delay* or *quality reduction* that has to be expected w.r.t. the process P_2 is waiting for. Second, aborting or partially rolling back P_1 and P_2 in case of a failure of P_1 may not always be appropriate as both processes may be able to continue after a dynamic adaptation.

In the context of the VIRTUAL ENTERPRISE CO-ORDINATOR (VEC), [LUDWIG & WHITTINGHAM 1999] introduce so-called agreement-driven gateways for cross-organizational workflow management. The related failure management approach [LUDWIG 1999] deals with unexpected *terminations* of workflows or workflow activities. In case of such a termination, the gateway protocol informs cooperating workflows about the termination reason and the state of the failed workflow or activity before termination. Furthermore, cooperating workflows may be informed

5. Opera = Open Process Engine for Reliable Activities

about a modification of an already agreed-on *price* for the service or product that is going to be provided by the workflow affected by the failure. As an example, the author describes a mail order workflow where the activity delivering the ordered product terminates because the receiver address turns out to be wrong. Thus, an increased price may have to be paid by the receiver as the mail order workflow has to figure out the right address and has to send the product again. To determine the price modification due to a termination, exception rules can be assigned to agreements specifying under which termination circumstances which price modification shall be applied.

Though the described approach is a first step in “measuring” the consequences of a failure for cooperating workflows by determining price modifications, it shows several limitations:

- First, it does not cover failures *not* leading to a workflow or activity termination but for example to the dynamic adding or postponing of some activities.
- Second, mapping the consequence of a failure situation to price modifications is quite restrictive. In e-commerce applications often failure-caused *temporal delays* are at least as important as price modifications. Furthermore, in many non-commercial cooperation scenarios the consequences of failures cannot only be measured in terms of time and money. For example, in medical care a lot of parameters exist which measure the effectiveness of a therapy in terms of drug dosages and disease remission. These parameters usually are also influenced by workflow failures.

In the context of the CMI⁶ project [GEORGAKOPOULOS ET AL. 1999 B], a crisis mitigation approach for collaborating processes is proposed in [GEORGAKOPOULOS ET AL. 2000]. In the terminology of the authors, a *crisis* is a situation which has to be controlled as soon as possible but for which the exact course is unknown and unpredictable. Crisis examples may include airplane crashes or the outbreak of an epidemic that has to be managed cooperatively by health care organizations. To deal with such crises, the authors assume so-called *crisis mitigation processes* specified in advance by the responsible organizations to avoid chaotic response and to increase mitigation effectiveness. Such a mitigation process then has to be adapted w.r.t. the specific requirements of the current crisis, e.g., by creating new activities, roles and task forces as needed to deal with the current crisis demands, and by delegating responsibilities to process participants and task forces. Furthermore, as for some activities both the exact execution *position* within the process specification and the *number* of executions may not be known beforehand, so-called *windows of opportunity* can be defined. At execution time, such a window of opportunity has to be refined w.r.t. its temporal duration and the number of activity executions. This crisis-oriented adaptation of processes is termed *process escalation*.

Technically, process escalation is mainly achieved by a *late modeling* approach (see Section 2.4.1), i.e., the mitigation process specifications consist of an arbitrary number of *placeholders* that have to be filled out at execution time. In particular, *activity placeholders* and so-called *repeated optional dependencies* allow both expert participants and co-ordinators to escalate the process.

6. CMI = Collaboration Management Infrastructure

Activity placeholders allow for activities whose concrete types are unknown or intentionally left open at process specification time. A repeated optional dependency consists of a *repeated option creator* and a *terminator*. The repeated option creator specifies that an activity *A* may be executed zero or more times. The time and number of executions is then determined dynamically by the process participants assigned to *A*. The repeated option terminator limits the time span within which *A* can be executed. For example, the terminator could specify that the executions of *A* have to be terminated when another activity *B* is first started. The process designer may also specify an upper bound on the total number of executions that can be performed. Activities constrained by such repeated optional dependencies cannot have outgoing dependencies to the rest of the activities in the process.

Though the proposed escalation model provides useful modeling primitives to adapt processes in a crisis-oriented manner, it shows the general limitations of late-modeling approaches (see discussion in Section 2.4.1). In particular, asynchronous events that have not already been reflected by placeholders cannot be handled. Furthermore, although the authors address *collaborative* process management, the implications a process escalation may have for *other collaborating* processes are not explicitly addressed. For example, if a crisis expert dynamically determines that in a process *P* an activity should be repeated 3 times instead of only once, the *temporal* implications for another process waiting for results from *P* are not determined and handled.

In [KLINGEMANN ET AL. 1999] a technique is described that allows to derive a model of external services in cross-organizational workflows from the *externally observable* service behavior. The approach is based on continuous-time Markov chains that can be incrementally constructed from the log of the past executions of services. This allows the service requester to build up an external model of services and to assess their quality without compromising the autonomy of the service providers. However, though Markov chains could be considered for requirement 3 (*Handling of Inter-Workflow Implications of Control Flow Failures*) by *stochastically* deriving which implications have to be expected in case of a failure, there usage for AGENTWORK is limited. This is mainly because the Markov chain property (i.e., a step only depends on the last step before) usually is not valid in medical domains, as diagnostic or therapeutic steps often depend on *several* predecessor steps (or even on the whole treatment chronology).

2.4.3 Summary

We summarize our discussion of approaches from the fields of adaptive and cooperative workflow management in Table 2-3. We emphasize that a “No” entry in the table does not mean that there is no support at all w.r.t. the particular requirement. Rather, “No” means that to the best of our knowledge the support is not explicit and sufficient enough for the purposes of this thesis.

The requirements not sufficiently supported by current approaches will be addressed in this thesis. Related work from the field of *temporal* workflows, such as deadline management and algorithms estimating a workflow’s duration [ADAM ET AL. 1998, EDER ET AL. 1999 A, MARJANOVIC &

Requirements for the Handling of Control Flow Failures	Supported	Representative Approaches and Remarks
1.1 High Semantic Level of Event and Control Action Representation	(Yes)	ADEPT _{FLEX} (w.r.t. actions)
1.2 Temporal Structure of Events	Yes	CHIMERA-EXC (e.g., deadline violation)
1.3 Temporal Structure of Control Actions	No	
1.4 Integrity of Failure Rules	No	
1.5 Authorization of Control Actions	Yes	ADEPT _{FLEX}
2.1 Workflow Abortion and Suspension	Yes	Advanced transaction models (2.2)
2.2 Support of Reactive Adaptation	Yes	CHIMERA-EXC, MOKASSIN
2.3 Support of Predictive Adaptation	No	
2.4 Consideration of Data Flow Implications	Yes	ADEPT _{FLEX}
2.5 Consistency of Adapted Workflows	Yes	ADEPT _{FLEX} , WORKFLOW NETS
2.6 Efficiency of Adaptation	Yes	ADEPT _{FLEX}
3.1 Determination of Temporal Implications	(Yes)	ADEPT _{FLEX} (but only execution durations specified at definition time; no duration measurements)
3.2 Determination of Qualitative Implications	(No)	VEC (supports only implications for product prices)

Table 2-3: Support of research approaches from adaptive and cooperative workflow management

ORLOWSKA 1999] will be discussed later on in Chapter 6 (*Workflow Duration Estimation*) after the specific temporal aspects of our adaptation approach have become clearer.

2.5 Artificial Intelligence Approaches

In this section, we discuss approaches from the field of artificial intelligence which are relevant for the handling of control flow failures in workflow management systems, namely approaches from the fields of planning (2.5.1), cooperative agents (2.5.2), and temporal reasoning (2.5.3).

2.5.1 Planning

In the terminology of artificial intelligence, planning is the task of choosing and ordering a sequence of actions (steps) needed to achieve a set of objectives [MCDERMOTT & HENDLER 1995, AYLETT ET AL. 2000]. A planning problem is usually defined by a domain model and by two states of that model: the initial state and the goal state. The domain model describes the objects in a domain (e.g., the available drugs in a medical domain), the actions that can be performed with these

objects (e.g., the administration of a drug), and the constraints on these actions (e.g., the side-effects that have to be considered when administering a drug). Actions are normally defined by so-called *planning operators*. The initial state describes the state of the domain immediately before any actions have been carried out, while the goal state describes the facts which should be true after the plan has been completed (e.g., the remission degree of a tumor). From the workflow point of view, planning can be viewed as a task that constructs a control flow over activities to meet a given goal. Consequently, there have been some efforts to apply planning techniques to workflow management [BECKSTEIN & KLAUSNER 1999B, SINGH & HUHS 1994, LIU & CONRADI 1993].

The necessity to *adapt* plans to cope with new situations has early been identified by several authors in this field [BROVERMAN & CROFT 1987, WALTON ET AL. 1987]. Thus, as a plan roughly corresponds to the control flow of a workflow, it makes sense to discuss plan adaptation approaches to see whether they contribute to the handling of control flow failures. For this, we first sketch principal planning techniques. Second, we discuss representative plan adaptation approaches.

Planning techniques are typically classified into *linear* and *non-linear* planning [PUPPE 1993]. Linear planning assumes that the planning operators are *totally* ordered with the consequence that only operator sequences can be performed. In contrast to this, for nonlinear planning the operators are only *partially* ordered to allow other execution modes such as parallel operator execution as well. However, a limitation of nonlinear planning is that many problems are NP-hard [CHAPMAN 1987], such as the problem of determining whether a proposition is necessarily true in a nonlinear plan for which the action representation allows conditional actions. A linear planning technique has been implemented within the system STRIPS [FIKES & NILSSON 1971]. The system SIPE [WILKINS 1988] is an example for a nonlinear planning system.

Another way to classify planning techniques is to distinguish between *non-hierarchical* and *hierarchical* planning [HERTZBERG 1989]. Non-hierarchical planning tries to construct a plan in one turn. In contrast to this, hierarchical planning performs plan construction in different abstraction steps to reduce complexity. First a rough plan is created, which is then refined step by step. This corresponds to hierarchical workflow modeling where activities are refined by subworkflows. Both for non-hierarchical and hierarchical planning, linear or non-linear planning techniques can be used.

For other planning techniques such as *real time* planning [ATKINS ET AL. 1999], planning *under uncertainty* [DEJONG & BENNET 1997], *agent-based* planning [DRABBLE & TATE 1995], or planning under *resource constraints* [KÖHLER 1998], we refer to the respective literature.

Despite the similarities between planning techniques and workflow management, there are substantial differences that have to be considered when discussing the usage of planning techniques for control flow failure handling [REICHERT 2000]. First of all, to model conditional paths, complex extensions become necessary for the above planning techniques (*conditional* planning). Second, many planning techniques do not support loops which are essential for workflow management. Third, most planning systems do not explicitly consider data flow aspects as they assume that all relevant data are globally available. As a direct consequence, the integration of planning systems into real-world environments is difficult.

We now discuss two representative plan adaptation approaches, namely those of the planning systems CHEF and CPEF.

CHEF [HAMMOND 1990] is a hierarchical planning system that is based on a case-based reasoning approach [KOLODNER ET AL. 1985], i.e., it takes existing plans out of its memory and incrementally adjusts and refines them w.r.t. the current planning goals. In our context, the most important module of CHEF is the so-called REPAIRER. This module is able to detect when a plan has failed, to repair the plan, and to store and index the repaired plan in its plan base to use it later on when constructing new plans.

To detect failures, REPAIRER uses two principal criteria: First, a plan fails if the goal assigned to the plan is violated. Second, it fails if a precondition of an action is violated. When such a failure has been detected, REPAIRER attempts to repair the plan by removing, adding or replacing actions. For this purpose it uses a base of rules and constraints stating when an action has to be removed, added, or replaced. For example, if a violated precondition led to the plan failure, REPAIRER would attempt to replace the violating action by another action that has a precondition not being violated and that does not violate the plan's goal.

CPEF (Continuous Planning and Execution Framework) is a planning system to support the generation, execution, monitoring, and repairing of plans in unpredictable and dynamic environments such as military campaign planning [MYERS 1998]. In contrast to CHEF [HAMMOND 1990], the proposed framework does not restrict plan failures to the violations of single preconditions and goals. Rather, also so-called *aggregate failures* are supported reflecting that often *multiple* precondition or goal violations together cause a plan failure. For example, air campaign plans often include extra missions beyond what is required to satisfy the military goals in order to improve the likelihood of success. Thus if only a few of these extra mission goals are violated this does not already induce the failure of the whole plan.

Plan repair in CPEF is performed as so-called *asynchronous run-time replanning* [WILKINS ET AL. 1995] based on the nonlinear SIPE planner [WILKINS 1988]. This means that plan execution continues with those branches of the plan that are not affected by the failure while the other affected branches are repaired. This mode of operation contrasts with synchronous replanning where plan execution is halted while the plan is repaired (as, for example, in CHEF). The authors view asynchronous replanning as essential for many domains as it would be infeasible to halt execution while only some parts of the plan are adapted. Replanning itself is then performed by applying plan adaptation rules which remove or insert actions or change goals, similar to the approach described in [HAMMOND 1990]. Because of the asynchronous replanning mode, the system also continuously reconciles the adapted plan with the state to which plan execution has progressed during replanning.

Although the above-mentioned approaches provide a flexible failure handling for a broad range of planning scenarios (see, for example, [CLYMER 1993] for a discussion of CHEF), their usage in our specific context is limited:

- First, the *temporal* dimension of plan failures and repairs is not supported sufficiently. Though approaches such as CHEF and CPEF support temporal constraints that induce a plan failure in case of their violation, they are not able to repair a plan for a specified temporal window such as “for the next seven days” or “until a termination condition *C* holds”. However, this is necessary, for example, in many medical contexts. In particular, *predictive* adaptations based on temporal estimations are not addressed sufficiently. Though the temporal dimension of plan actions has early been identified as important [MCDERMOTT 1982] and several plan duration estimation algorithms exist [BLUM & FURST 1997, SCHWALB & DECHTER 1997], the latter do not have explicitly been used for predictive plan adaptation.
- Second, *cyclic* processes are not considered. Thus, the plan repair approaches cannot be used to adapt a control flow containing a loop. In particular, the approaches do not address the problem for how many loop iterations an adaptation shall be valid.
- Third, the integrity of rule bases for plan adaptation is not explicitly addressed. Though several approaches have been proposed in artificial intelligence concerning the rule base integrity in general [SUWA ET AL. 1982, MESEGUER 1992, ROANES-LOZANO ET AL. 2000], these approaches have not specifically been used to avoid incompatible plan adaptation rules.
- Fourth, the above approaches usually restrict data flow aspects to the information exchange between the components performing the plan adaptation. The consequences a plan adaptation may have for the data flow between the plan *actions* themselves are not considered.
- Fifth, due to the inherent complexity especially of nonlinear planning, their practical usage for real-world applications is limited.

Several other approaches have been suggested to support workflow flexibility and adaptation by using planning techniques [DELLEN ET AL. 1997, BECKSTEIN & KLAUSNER 1999B]. However, none of these approaches suggests technical solutions going beyond those already provided by CHEF and CPEF, so that we omit details here.

2.5.2 Cooperative Agents

Recently, also techniques from the field of *cooperative agents* [KRAUS 1997, LESSER 1998] have been applied to process and workflow management [HALL & SHAHMEHRI 1996, MERZ ET AL. 1996, JENNINGS ET AL. 2000, KLEIN & DELLAROCAS 2000 B]. Concerning the term *agent* we refer to [WOOLDRIDGE 1997] who characterizes an agent as “an encapsulated computer system that is situated in some environment and that is capable of flexible, autonomous action in that environment in order to meet its design objectives.”⁷ However, only a few approaches address failure handling for agent-based workflow management.

For example, Jennings et al. describe the project ADEPT⁸ [JENNINGS ET AL. 2000] where the main

7. For the ongoing debate what exactly characterizes agents we refer to [JENNINGS 2000, FRANKLIN & GRAESSER 1997].

components are interacting, autonomous agents being responsible for performing particular activities. The term *service* is used to denote a manual or automated activity that an agent can manage and that forms a conceptual unit in a business process, such as designing a software module or reviewing a paper for a scientific journal. A resource is anything that is needed to perform a service (e.g., material, time, or money). If an agent needs a service that is managed by another agent during process execution, this requires both agents to come to a mutually acceptable agreement about the terms and conditions under which the desired service will be performed. In the terminology of the authors, such contracts are called *service level agreements* (SLAs). The mechanism for making SLAs is termed *inter-agent negotiation* – a process in which parties verbalize possibly conflicting demands and then move towards agreement by making concessions or by searching for new alternatives [MÜLLER 1996].

The so-called *situation assessment module* (SAM) is responsible for assessing and monitoring an agent's ability to meet the SLAs it has already agreed to and the potential SLAs which it may agree to in the future. This involves *scheduling* and *exception handling*. The former involves maintaining a record of the availability of the agent's resources which are used to determine whether SLAs can be met or whether new SLAs can be accepted. The exception handler receives exception reports during service execution such as "*service may fail*", "*service has failed*", or "*no SLA in place*", and decides upon the appropriate response. For example, if a service is delayed the SAM may decide to locally reschedule it, to renegotiate its SLA, or to terminate it altogether.

The specific contribution of ADEPT is that it models business processes in terms of *services*, *agents*, *(re-)negotiations*, and *service failures*, and therefore on a high level of application-oriented abstraction. However, concerning failure handling ADEPT does not provide any *technical* details *how* failures are detected and *how* they are resolved in terms of structural process adaptations. For example, no formal criteria or estimation algorithms are suggested which could derive that a service "may fail". Thus, ADEPT can be viewed as an agent-oriented protocol that could be used *on top of* a workflow management system and the failure handling mechanism as addressed in this thesis. Furthermore, the emphasis on *negotiation* may be suitable for e-commerce applications but it is of limited use for workflow management in cooperative health care and other non-commercial workflow application scenarios.

In [KLEIN & DELLAROCAS 2000 B, KLEIN & DELLAROCAS 1999], an agent-based workflow system is described consisting first of so-called *problem solving agents* which focus on managing their own "normal" problem, such as the execution of some manufacturing and delivery workflow in an e-commerce scenario. Second, *exception handling agents* concentrate on detecting and resolving exceptions during workflow execution. The exception handling is based on *heuristic classification* [CLANCEY 1985] known also from the field of medical diagnosis. In contrast to traditional rule-based exception handling approaches, heuristic classification supports data *abstraction* and intermediately allows *competitive diagnoses*. Technically, this is achieved by arranging exception

8. ADVANCED DECISION ENVIRONMENT FOR PROCESS TASKS (not related to ADEPT_{FLEX} described in [REICHERT & DADAM 1998])

types in a taxonomy ranging from highly general failure modes at the top to more specific ones at the bottom. Every exception type includes a set of defining characteristics that need to be true in order to make the diagnosis assigned to it potentially applicable to the current situation. For example, in a manufacturing workflow a time-out exception may occur indicating that some item needed for assembling the product is missing. Competitive diagnoses obtained from the taxonomy may include that the responsible (human) agent is unavailable (e.g., ill), that the item has been misrouted or that the production of the item itself has been delayed. To identify the right diagnosis, a user then has to answer a specific set of questions in order to narrow down the exception diagnosis. Alternatively, if the needed information is available in a database, diagnosis elimination can also be automated.

Once an exception has been detected and diagnosed, the system attempts to resolve it. This is achieved by selecting and instantiating one of the generic exception resolution strategies that are associated with the proposed diagnosis. These strategies are processes like any other and are annotated with attributes defining preconditions for their execution. Strategies suggested by the authors for example include:

- If a highly serial process is operating too slowly to meet an impending deadline, then partial results already available are released to allow later activities to start earlier (termed *pipelining*), or execution is parallelized to meet the deadline.
- If an agent may be late in producing a time-critical output, then it is checked whether the consumer agent will accept a less accurate output in exchange for a quicker response.

The strength of the approach is the application of a powerful, well-proved diagnostic method (i.e., heuristic classification) for the detection and handling of exceptions. This method is more appropriate than traditional ECA rules as it supports, for example, data abstraction and competitive diagnoses. However, there are several limitations making the approach unsuitable for the problems addressed by this thesis:

- Though heuristic classification is applied to workflow management for the purposes of exception handling, the authors do not really adapt it in a *workflow-oriented* manner. In particular, especially the resolution strategies are only formulated on a relatively abstract level, and not in terms of dropping, adding or re-ordering workflow activities. Thus, for a workflow the *structural* consequences of exception resolution in terms of control and data flow adaptations remain unclear.
- Furthermore, there is no automated derivation about the implications an exception resolution may have for cooperating workflows. For example, if a producer agent may be late in producing a time-critical output, one resolution strategy mentioned above consists of checking whether the consumer agent will accept a less accurate output in exchange for a quicker response. However, this usually is only one of the later steps during the exception handling process. The authors do not discuss any strategies determining *predictively* whether the workflow to which some exception occurred will be unable to meet a deadline in the future. As workflows usually consist of conditional branches and loops, this is not a trivial problem.

2.5.3 Temporal Reasoning

According to requirements 1.2 (*Temporal Structure of Events*) and 1.3 (*Temporal Structure of Control Actions*), the representation and processing of temporal structures plays an important role for the handling of control flow failures. Thus, we inspect work from the field of *temporal reasoning* which addresses the formalization of time and the processing of temporal information [VILA 1994]. Typically, a temporal reasoning framework consists of two main components:

- *A temporal ontology*: The ontology describes the “structure of time“ by specifying the temporal primitives (e.g., points in time, intervals etc.), and the temporal relations that exist between them. Often, temporal ontologies are classified into interval-based [ALLEN 1984] and point-based ontologies [DELGRANDE ET AL. 1999].
- *A temporal reasoning system*: This component allows one to determine the truth of any temporal assertion expressed in terms of the underlying temporal ontology.

Most temporal reasoning systems are based on logics due to their well-founded semantics and computational power [SCHÖNING 1989]. Three principal methods of logic-based temporal reasoning can be identified, namely first-order logics with temporal arguments, modal temporal logics, and reified temporal logics [VILA 1994, MA & KNIGHT 2001]:

- *First-order logics with temporal arguments* [HAUGH 1987]: This method simply consists of representing time just as another parameter in a first-order predicate calculus. Logical functions and predicates are extended with additional temporal arguments denoting the particular time at which they have to be interpreted. The advantage of this method is that it can use the well-known proof theory of first-order logics without having to extend it for temporal purposes. Its main shortcoming is its low expressive power. In particular, as time is just another parameter between others, the reasoning system cannot distinguish between temporal and non-temporal assertions.
- *Modal temporal logics* [GABBAY 1987]: This method is an extension of the propositional or predicate calculus with so-called modal temporal operators [PRIOR 1955] such as (ϕ being a formula):

$F \phi$	meaning	ϕ is true in some future time, or
$P \phi$	meaning	ϕ is true in some past time.

Concerning expressiveness, modal temporal logics are better than first-order logics with temporal arguments [VILA 1994], so that they are preferred for complex tasks such as natural language processing and program verification. However, the complexity of their theorem proving is higher than that of first-order logics and therefore makes modal temporal logics less attractive for many application classes.

- *Reified temporal logics* [MCDERMOTT 1982]: This method can be understood as an attempt of achieving a higher expressive power than first-order logics with temporal arguments while staying in the proof theory of first-order logics. This is done by moving into a temporal meta-

language where a formula of an initial first-object logic language becomes a term – namely a propositional term – in the temporal meta-language [VILA 1994]. In particular, the “truth predicates” of the temporal meta-language take as arguments a formula in the initial first-object logic language and an expression denoting a temporal object, such as

Holds(critical-hemato-status(“Bob Miller”), 17 Aug 2001) (i)

with *Holds* being a truth predicate of the temporal meta-language, and *critical-hemato-status* being a predicate of the initial first-order logic language. The assigned temporal object – e.g., *17 Aug 2001* in (i) – is often also termed as the *valid time* of an assertion. As this mechanism on one side provides an expressiveness that is sufficient for many application classes, and on the other side keeps the well-known proof theory of first-order logics, reified temporal logics have been favored in temporal reasoning by many authors [MA & KNIGHT 2001]. Nevertheless, an open issue is characterizing efficient specialized inference methods for first-order reified languages [VILA 1994].

Due to the qualities of a reified temporal logic in contrast to a first-order logic with temporal arguments or a modal temporal logic, several approaches have been proposed to formalize *events* and their temporal structure (requirement 1.2.) on the basis of such a reified temporal logic (e.g., [MOTAKIS & ZANIOLO 1997 A, DINN ET AL. 1999]). For example, in [MOTAKIS & ZANIOLO 1997 A] the temporal language TREPL is introduced which allows to define composite events and which provides temporal aggregation constructs to express time series such as that for the daily closing prices for a fixed set of stocks. Formally, TREPL is based on a reified version of the logic-based programming language DATALOG [CERI ET AL. 1989]. However, a limitation of the approach is that no rule templates are provided to define application-specific event aggregations.

As the aim of this thesis is *not* to provide a general-purpose event specification and processing approach but to cope with events in a way suitable for control flow failure handling, we do not discuss further event-oriented temporal reasoning approaches here, but refer to Chapter 4 (*Data and Rule Definition with ActiveTFL*). There, existing temporal reasoning approaches based on reified temporal logics will be adapted for the event-handling purposes of this thesis.

Concerning control actions (requirement 1.3.), there are no specific approaches from the field of temporal reasoning addressing this problem for workflow failure handling. However, as we will see in Chapter 4 (*Data and Rule Definition with ActiveTFL*), due to their expressiveness reified temporal logics provide the necessary aspects to specify control actions and their temporal structure.

For advanced temporal reasoning approaches that are beyond the scope of this thesis, we refer to the literature, e.g., to [STURM & WOLTER 2002, CERRITO & MAYER 1998, VAN BEEK & MANCHAK 1996].

2.5.4 Summary

We summarize our discussion of approaches from the field of artificial intelligence in Table 2-4. Again, we emphasize that a “No” only means that the support is not explicit and sufficient enough

Requirements for the Handling of Control Flow Failures	Supported	Representative Approaches and Remarks
1.1 High Semantic Level of Event and Control Action Representation	Yes	ADEPT [JENNINGS ET AL. 2000]
1.2 Temporal Structure of Events	Yes	CHEF, temporal reasoning approaches
1.3 Temporal Structure of Control Actions	(No)	Not specifically for failure handling
1.4 Integrity of Failure Rules	No	Existing rule integrity approaches not used for failure handling
1.5 Authorization of Control Actions	No	
2.1 Workflow Abortion and Suspension	No	
2.2 Support of Reactive Adaptation	Yes	CHEF, CMEP, ADEPT
2.3 Support of Predictive Adaptation	No	
2.4 Consideration of Data Flow Implications	No	
2.5 Consistency of Adapted Workflows	Yes	CHEF, CMEP
2.6 Efficiency of Adaptation	No	
3.1 Determination of Temporal Implications	(No)	No explicit use of algorithms estimating plan durations for failure handling
3.2 Determination of Qualitative Implications	(No)	Only in terms of goal violations

Table 2-4: Support of research approaches from artificial intelligence.

for the specific purposes of this thesis.

2.6 Chapter Summary

In this chapter we discussed approaches from commercial workflow management systems, advanced transactions models, exception handling in programming languages, adaptive and cooperative workflow management, and artificial intelligence. The main result has been that especially the requirements 1.3 (*Temporal Structure of Control Actions*), 2.3 (*Support of Predictive Adaptation*) and 3 (*Handling of Inter-Workflow Implications of Control Flow Failures*) are not supported sufficiently. Therefore, this thesis will focus on these requirements.

Chapter Summary

This chapter gives an overview of the workflow management system AGENTWORK. The principal goal of AGENTWORK is to support the requirements discussed in Chapter 1 and summarized in Table 2-1, due to the limitations of current workflow systems. In particular, the support of predictive adaptation is one of the central goals of AGENTWORK.

The purpose of this overview is to motivate and describe central design decisions of AGENTWORK before going into the technical details in the following chapters. In particular, we show that *temporal* aspects play an important role at nearly all steps during the handling of control flow failures so that it becomes clear why a temporal logic has been selected as primary specification language for AGENTWORK. The chapter is organized as follows: In Section 3.1 we briefly describe the three architectural layers of AGENTWORK and their components. Section 3.2 sketches the workflow definition and execution model. Section 3.3 introduces rules for control flow failures. These rules play a central role as they identify failure events and trigger control actions stating how to handle such failure events. Section 3.4 sketches how these control actions are transformed into structural workflow adaptations. A summary and discussion in Section 3.5 completes the chapter.

3.1 Layers and Components of AGENTWORK

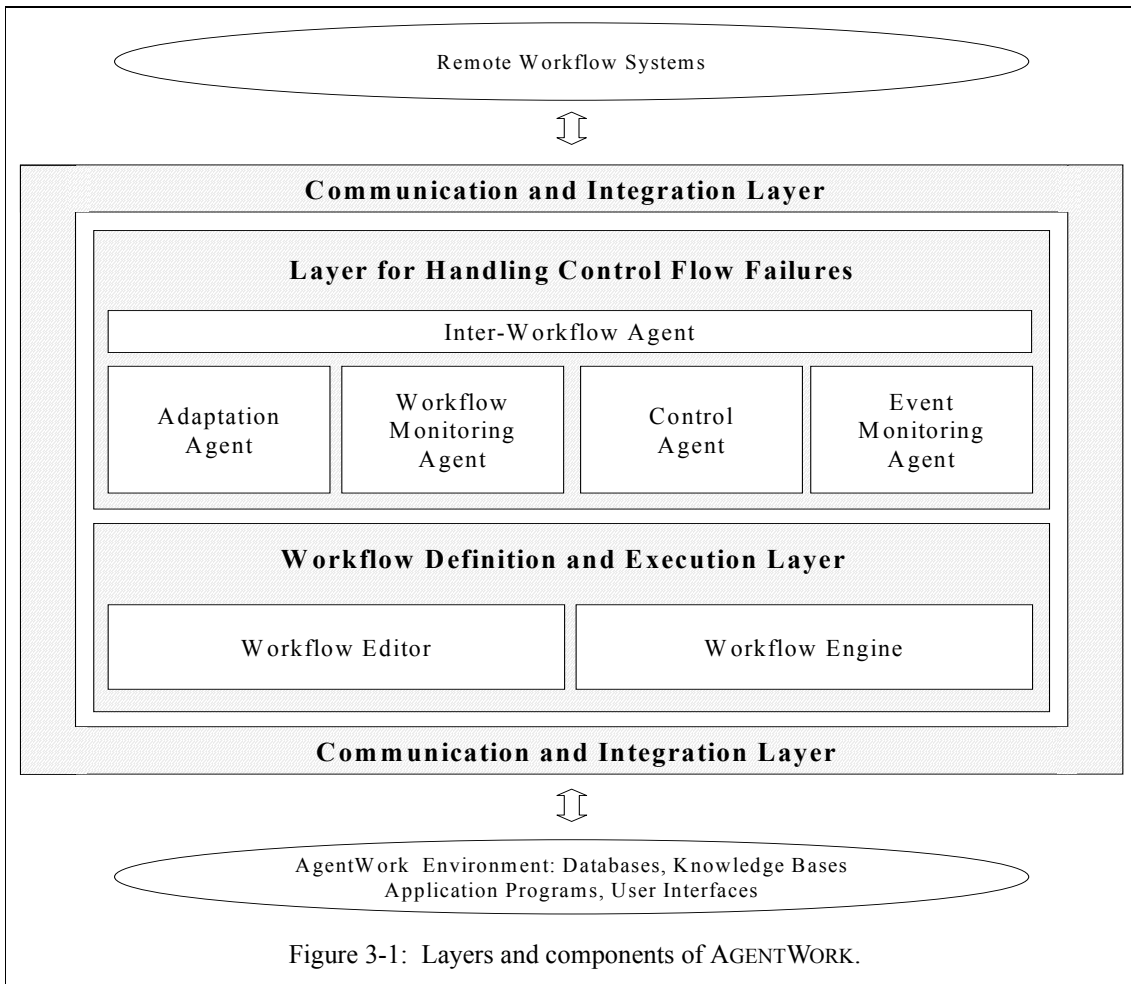
To clearly separate the basic workflow management from failure handling aspects and communication, AGENTWORK consists of three architectural layers: The workflow definition and execution layer, the layer for handling control flow failures, and the communication and integration layer (Figure 3-1).

Workflow Definition and Execution Layer

This layer provides components for the definition and execution of workflows. A workflow editor and a workflow engine form the main components corresponding to the core of a typical workflow management system as sketched in Chapter 1. However, this layer differs from other workflow management systems as it, for example, allows to suspend or adapt workflows which are currently being executed.

Layer for Handling Control Flow Failures

This layer implements the main concepts of this thesis. It provides five agents for the handling of control flow failures (see Definition 1.1 in Chapter 1). The *event monitoring agent* decides which application events occurring somewhere in the workflow environment constitute relevant failure



events. The *control agent* determines which running workflows are affected by such a failure event. In case of a *global* failure event affecting a workflow as a whole the control agent instructs the workflow definition and execution layer to abort or suspend the workflow. In case of a *local* failure event affecting only some activities of a workflow the *adaptation agent* is instructed to adapt the workflow. This adaptation agent, for example, removes or inserts activities so that the workflow can better cope with the new situation caused by the failure event. The so-called *workflow monitoring agent* is additionally involved to check whether the adaptation assumptions of the adaptation agent are matched when the adapted workflow is executed. This is necessary, as the adaptation agent – depending on the type of control flow failure – *estimates* which parts of a workflow will be executed during a given temporal interval. Finally, the *inter-workflow agent* determines whether a control flow failure occurring to a workflow has any implications for other workflows cooperating with this workflow. These cooperating workflows may be executed by the same workflow engine as the workflow to which the failure occurred, or may belong to external workflow systems in other departments or even other enterprises or hospitals.

The components of this layer are called *agents* because they have several properties, that are associated with agent-oriented modeling and programming, such as “*intelligence*“, *autonomy* and *cooperation* [WOOLDRIDGE & JENNINGS 1995 A, FRANKLIN & GRAESSER 1997]. However, the thesis does not aim at contributing to agent technology *research*. For the latter, we refer to the literature (e.g., [MÜLLER ET AL. 1999 B, MEYER & SCHOBENS 1999, BRADSHAW ET AL. 1999]).

Communication and Integration Layer

This layer manages the communication between the workflow definition and execution layer and the layer for handling control flow failures on one side, and the AGENTWORK environment and remote workflow systems on the other side. To address requirement 4 (*Handling of Data and Event Distribution and Heterogeneity*) discussed in Chapter 1, the communication and integration layer is based on the middleware CORBA [BAKER 1997]. As it has been implemented in a straightforward manner and does not address any aspects of interoperability research, we refer to Chapter 11 (*Implementation Issues*) for further details.

3.2 Workflow Definition and Execution

We now sketch how workflows are defined and executed in AGENTWORK. This covers the global AGENTWORK data schema, and the model of workflow definitions and workflow instances.

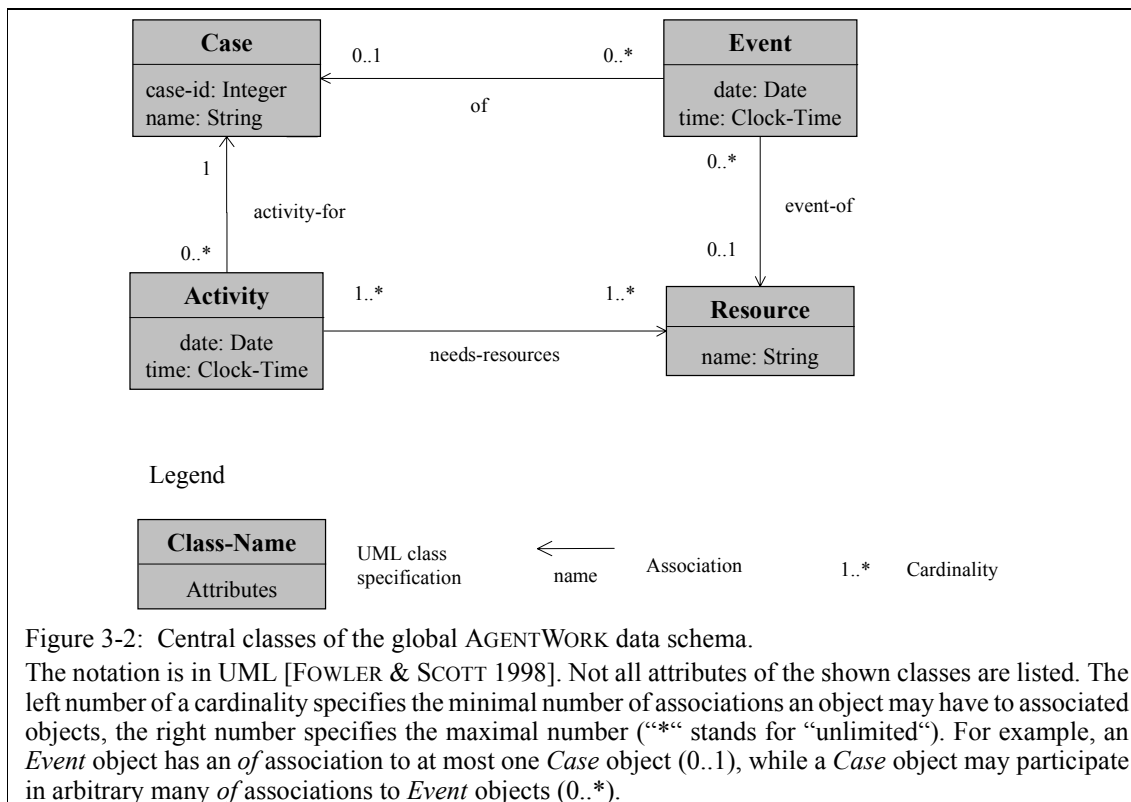
3.2.1 Global Data Schema

In AGENTWORK all relevant elements such as activities and events are specified on the basis of an object-oriented *global data schema* on a high level of abstraction. This supports user interaction and achieves transparency for the agents of the layer for handling control flow failures w.r.t. the distribution and heterogeneity of application data (requirement 1.1: High Semantic Level of Event and Control Action Representation). The dependencies between this global data schema and the local

data sources are managed by the communication and integration layer mentioned above.

In this global data schema, the classes *Case*, *Resource*, *Activity*, and *Event* play a central role for the process of handling control flow failures (Figure 3-2). This is because control flow failures first are viewed as an inadequacy of a workflow's activity set, and second are induced by events occurring to cases or resources (1.2.2). These classes are sketched now in an informal manner:

- Class *Case*: An object of this class represents a person or institution for which an enterprise or organization provides its services and thus executes its workflows. In a medical application, the *patient* is the typical case. A typical case for insurance business or banking is the *customer*.
- Class *Activity*: An *Activity* object represents something that is executed for a case (*activity-for* association between *Activity* and *Case*). A single activity instance is executed for exactly one case, but for a case an arbitrary number of activities may be executed. In a medical application, a typical activity is a diagnostic examination or a drug administration.
- Class *Resource*: A *Resource* object represents someone or something needed to execute an activity (*needed-to-execute* association between *Resource* and *Activity*). Such a resource may either be a person, an application program or some piece of equipment. At least one resource is



needed to execute a single activity instance, and a resource may be involved within the execution of an arbitrary number of activities. In a medical application, a personnel resource could be a physician or nurse performing treatment activities for the patient. An example of a piece of equipment is a computer tomograph.

- Class *Event*: Objects of the class *Event* represent anything that happens w.r.t. a case or a resource. In AGENTWORK an arbitrary number of events may happen w.r.t. one case or resource, but a particular event happens either to exactly one case or exactly one resource (*of* association between *Event* and *Case* respective *Resource*). In a medical application, a case-related event could be that the patient has a decreased leukocyte count. A case-related event in insurance business could be that a customer causes a car accident for which the insurance company has to be pay the damage. A resource-related event could be that the computer tomograph of a ward gets broken. We recall from Chapter 1 that both case- and resource related events may require that a workflow has to be adapted dynamically.

Details about the global data schema can be found in Chapter 4 (*Data and Rule Definition with ActiveTFL*).

3.2.2 Workflow Definitions

On the basis of the global data schema, workflows can be defined. Workflow activities are represented by so-called *activity nodes*. To an activity node, a so-called *activity definition* has to be assigned specifying which activity has to be executed when the activity node is reached. Such an activity definition has the general structure

activity-class[*attribute-value-set*]

where *activity-class* is an *Activity* subclass of the global data schema (Section 3.2.1), and *attribute-value-set* is a valuing of the attribute set of *activity-class*. An example for such an activity definition is (informal notation)

Drug-Administration[*drug* = “G-CSF“, *dosage* = 300, *unit* = µg, *type* = *injection*] (i)

specifying that a G-CSF injection with a dosage of 300 µg has to be administered (assuming that *Drug-Administration* is an *Activity* subclass). Furthermore, an activity definition consists of the specification which input objects are needed and which output objects are provided.

The *control flow* between activity nodes is specified by so-called *control nodes* and *control edges*. AGENTWORK provides control node types for the begin and the end of a workflow (node types START/END), for parallel execution (node types AND-SPLIT/AND-JOIN), for conditional branching (node types OR-SPLIT/OR-JOIN) and for loops (node types LOOP-START/LOOP-END). Control edges connect activity and control flow nodes to node sequences to specify which nodes have to be executed next when an activity or control flow node has been executed. As an additional control flow element, synchronization edges [REICHERT & DADAM 1998] are offered to synchronize nodes belonging to different execution paths.

The data flow is specified by so-called *data flow edges* which connect workflow nodes. AGENTWORK distinguishes two main types: *Internal* data flow edges specify the data flow between nodes within a workflow. For example, such an internal data flow edge could specify that within a node sequence an output object produced by one node is used as an input object of one of the following nodes. *External* data flow edges specify the data flow between activity nodes and external data sources such as databases or user interfaces. For example, let us assume that the output of an activity based on (i) consists of some report describing how much of the dosage actually has been administered to the patient and if there were any complications. Then, an external data flow between this activity node and a relational patient database could specify that this information has been inserted into some table of the patient database. For this, data manipulation or retrieval statements can be assigned to an external data flow edge.

Details about the AGENTWORK workflow definition model can be found in Chapter 5. This chapter will describe further definition principles of AGENTWORK, such as using symmetrical control flow blocks [REICHERT 2000]. Furthermore, it will be described in detail how the data flow between activities is specified, and why alternative workflow definition languages such as petri nets [AALST 1998] or state/activity charts [WODTKE & WEIKUM 1997] have not been selected.

3.2.3 Workflow Instances

At workflow execution time the workflow engine reads workflow definitions and executes them. As in most other workflow systems, such an executed workflow is termed *workflow instance* in AGENTWORK. In particular, for each activity node an *Activity* instance is created on the basis of the assigned activity definition. At a given point in time an arbitrary number of workflow instances based on the same workflow definition may be executed. In the following we may also say *workflow* instead of *workflow instance* when it is clear from the context that we mean a currently executed workflow and not a workflow definition.

An event may induce a control flow failure of a workflow instance *I* if this event occurs to a case for which *I* executes some or all of its activities (*activity-for* association between an activity and a case in Figure 3-2), or if the event occurs to a resource which is needed to execute activities of *I* (*needed-to-execute* association between a resource and an activity in Figure 3-2). The question under which circumstances events have implications for workflow instances is handled by *rules for control flow rules* which are sketched now.

3.3 Rules for Control Flow Failures

For a failure event, the relative point in time of its occurrence w.r.t. a particular position in a workflow definition typically is unknown in advance. Furthermore, the information needed to handle a control flow failure appropriately is also known often only at execution time. Thus, pre-modeling them by inserting for instance a conditional branching for the normal and the failure case is not appropriate. Rather, it is more flexible to define event-condition-action rules (ECA rules) [WIDOM & CERI 1996] which actively react on events, and to derive the implications for running workflows dynamically. Therefore, AGENTWORK uses such ECA rules for control flow failures. Their main

characteristic is that they trigger so-called *control actions* which declaratively state on a high level of abstraction what should be done to handle a control flow failure (see Definition 1.2). In particular, they do only make minimal assumptions first about the workflow definition language and second about how the activities are spread over different workflow definitions. This has the advantage that changing the workflow definition language or reorganizing activities and workflows has only minimal effects on ECA failure rules.

In the following, all ECA rules are given in an *informal* notation to focus on the principal aspects of ECA rules in AGENTWORK. The formal notation is a subject of Chapter 4.

3.3.1 ECA Rules with Control Actions

The **event-condition** part of an AGENTWORK ECA failure rule specifies which *event* constitutes a failure event under which *condition*. A typical example for an event-condition part is that the leukocyte count of a patient is less than a critical range, e.g.,

(ii)

WHEN *new hematological finding of patient P*
WITH *leukocyte count < 1000 #/mm³*

The **action** part states which *control action* has to be performed. AGENTWORK supports two main types of control actions:

Global control actions state that a workflow is not adequate anymore *as a whole*. For example, AGENTWORK supports the control action *abort(W,C)* which states that for case *C* (e.g., for a particular patient) workflows based on definition *W* have to be aborted. An example for an ECA rule with an abort control action is (with *W* being a workflow definition for chemotherapy support):

(iii)

WHEN *new hematological finding of patient P*
WITH *leukocyte count < 500 #/mm³*
THEN *abort(W,P)*

as leukocyte values less than 500 forbid to continue with a chemotherapy.

Local control actions state that only *some* activities of a workflow are not adequate anymore. Thus, the workflow can be continued but has to be adapted locally. For example, AGENTWORK supports – beside others – the two local control actions *drop(A,C)* and *add(A,C)*. They state that for case *C* an activity based on the activity definition *A* has to be dropped respectively added. An example for an ECA rule with such a local control action is

(iv)

WHEN *new hematological finding of patient P*
WITH *leukocyte count < 1000 #/mm³*
THEN *add(A,P)*

where *A* shall be the G-CSF drug administration of (i). This rule states that whenever a patient has a leukocyte count less than 1000 that then *one* activity of administering a G-CSF injection with a

dosage of 300 µg drug has to be added. A variation of the $add(A, P)$ control action is the control action

$$add-repetitively(A, d, P) \quad (v)$$

which states that A should be performed repetitively for patient P with period d .

Both for global and local control actions, two subtypes can be identified: **Case-related control actions** are triggered when an event occurs to a case (3.2.1). They state what has to be done with a workflow from the case point of view, e.g., which workflow has to be aborted for this case or which activities have to be dropped or added for this case. The control actions used in (iii)-(v) are examples for such case-related control actions. **Resource-related control actions** are triggered when an event occurs to a resource. For example, they state which workflow has to be suspended or which activities may have to be dropped if some resources are not available temporarily. An example for such a resource-related control actions is the following: Let CT denote the (only) computer tomograph of a ward. Then, an ECA failure rule with a resource-related (local) control action could be

$$\begin{aligned} \text{WHEN } & broken(CT) \quad // \text{ If computer tomograph is broken} \\ \text{THEN } & drop-activities-of(CT) \quad // \text{ drop all activities for which } CT \text{ is needed} \end{aligned} \quad (vi)$$

An ECA rule such as (vi) may look trivial. However, many workflow systems (in particular commercial systems) are not able to adapt their control flow even for simple resource limitations, such as a broken piece of equipment making it temporarily impossible to execute some activities.

With ECA rules such as (iv) and (vi), we can define a (*control flow*) *failure event* more precisely than it was done in Chapter 1:

Definition 3.1: (Control Flow) Failure Event

An application event is called a *control flow failure event* or simply *failure event* if it triggers at least one ECA rule with a global or local control action in its action part.

3.3.2 Composite Events

We recall from the discussion of requirement 1 (*Representation of Failure Events and Control Actions*) in Chapter 1 that it must be possible to express, for example, that a laboratory value must have been critical for several times to constitute a failure event. Based on approaches described in [CHAKRAVARTHY ET AL. 1994, MOTAKIS & ZANIOLO 1997 A], AGENTWORK therefore provides a *composite* event model. For example, the composite event type TIME-SERIES allows to specify the repetitive occurrence of an event: If A again is an activity as defined in (i) and EC the event-condition part specified in (ii), then the rule

$$\begin{aligned} \text{WHEN } & TIME-SERIES(EC, 3, \text{“one week”}) \text{ for } P \\ \text{THEN } & add(A, P) \end{aligned} \quad (vii)$$

specifies that an activity based on A has to be added when the leukocyte count of P has been less than 1000 (1st parameter in *TIME-SERIES*) for three times (2nd parameter) within one week (3rd parameter).

3.3.3 Activity Patterns

To give ECA failure rules more flexibility, AGENTWORK also allows to use activity *patterns* within the control actions. For example, instead of using an activity definition such as (i) which specifies all attribute values precisely, an activity pattern such as

$$B := \text{Drug-Administration}[drug = \text{"ETOPOSID"}] \quad (viii)$$

can be used in a rule as follows:

$$\begin{array}{ll} \text{WHEN} & \text{new hematological finding of patient } P \\ \text{WITH} & \text{leukocyte count} < 1000 \text{ \#}/\text{mm}^3 \\ \text{THEN} & \text{drop}(B, C). \end{array} \quad (ix)$$

This rule states that whenever a patient has a leukocyte count less than 1000 that then all activities administering the drug “ETOPOSID” – independently from their dosage or administration type – have to be dropped.

If A is an activity definition such as (i) or an activity pattern such as (viii), we call an activity based on A or matching A an **A-activity**, and a node executing an A -activity an **A-node**. In the following, we will only assume activity *patterns* in ECA rules, as patterns subsume activity definitions such as (i) (i.e., any activity definition can be viewed as a very restrictive activity pattern).

3.3.4 Valid Time

So far, the action part of a failure rule such as (ix) does not state whether the control action $\text{drop}(B, C)$ holds, for instance, only for “a moment” or “for ever” or for a number of days. As the discussion in Chapter 1 has shown, it is however essential for many applications that the temporal window during which a control action shall hold is specified precisely. Therefore, in AGENTWORK a so-called *valid time* can be assigned to a control action. The concept of valid time has been introduced in temporal databases to restrict the validity of a statement to some period of time [SNODGRASS 1999, CHOMICKI 1994]. In AGENTWORK, valid time is used to specify for which period of time a control action holds. AGENTWORK supports two principal valid time types for control actions:

Fixed valid time describes a period of time by an explicit duration specification, e.g.,

$$\text{THEN} \quad \underbrace{\text{drop}(B, P) \text{ during the next seven days}}_{\text{fixed duration}} \quad (x)$$

This line specifies that all activities based on B have to be dropped for the next seven days.

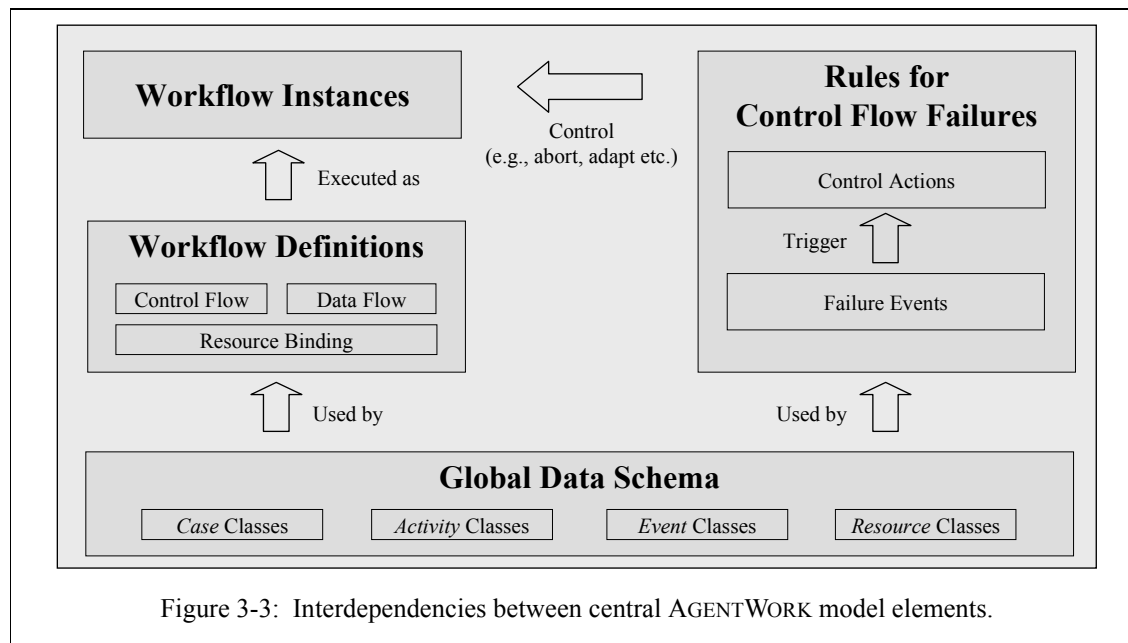
Conditional valid time describes a period of time by using *termination conditions*, e.g.,

$THEN \quad drop(B, P) \underbrace{until \text{leukocyte count of } P \text{ is higher than } 2500 \text{ again}}_{\text{termination condition}} \quad (xi)$

This specifies that all activities based on B have to be dropped for a patient P until the leukocyte count of P becomes higher than 2500. In particular, the duration such a conditional valid time is not known beforehand. A special type of termination condition is the condition “*until user cancels control action*” that is also supported. This is necessary, as for example the period of time during which a particular drug shall be dropped often cannot be specified at rule definition time. Rather, the physician dynamically wants to decide at execution time on the basis of the current findings when a control action is not valid anymore. However, for the ECA model the termination condition “*until user cancels control action*” is a condition among others and therefore can be handled like a condition such as the one in (xi).

Transaction time which stores *when* some information has been present in a data source [SNODGRASS 1999] is irrelevant for this thesis, so that we do not consider this type of time in the following.

Figure 3-3 summarizes the dependencies between the global data schema, workflow definitions, workflow instances, and rules for control flow failures as described in Sections 3.2 and 3.3. The



detailed ECA model and syntax of AGENTWORK – including the composite event model and the temporal model of ECA rules – will be described in Chapter 4. The complete listing and description of AGENTWORK supported control actions – including the problem of control action consistency and dependencies – will be given in Chapter 7.

3.4 Control Action Processing

We now sketch the principal way AGENTWORK processes triggered control actions. In this overview chapter, we concentrate on *local* control actions as they induce most of the complexity of AGENTWORK. Furthermore, we consider only that a *single* control action has been triggered. The question how *multiple* control actions and possible dependencies between them are handled is discussed in the chapters later on.

We first describe how AGENTWORK decides which adaptation strategy (reactive or predictive) shall be used when a control action is triggered (Section 3.4.1). Second, we sketch the principles of workflow estimation which is needed for predictive adaptation (Section 3.4.2). Third, we sketch how adaptation operators translate control actions into structural adaptations of the workflow (Section 3.4.3). Fourth, we discuss that such a structural adaptation may have some side-effects on a workflow that have to be controlled (Section 3.4.4). Fifth, we describe how a workflow that has been adapted by predictive adaptation is monitored to check whether the estimations are met by the actual execution (Section 3.4.5).

3.4.1 Strategy Selection

We recall from Chapter 1 (Section 1.3) that two principal adaptation strategies exist to handle control actions. *Reactive adaptation* handles a control flow failure only on immediate demand, e.g., drops a node that may not be executed anymore just before the node is scheduled for execution. In contrast to this, *predictive adaptation* adapts workflow parts that are farther away from the currently executed nodes. The main advantage of the latter strategy is that it gives the workflow users more time to prepare themselves w.r.t changed workflow definitions.

Thus, when a local control action such as *drop(A,C)* or *drop-activities-of(R)* (*R* being a resource) has been triggered by an ECA rule, AGENTWORK has to determine which adaptation strategy should be selected (Figure 3-4).

3.4.1.1 Predictive Adaptation

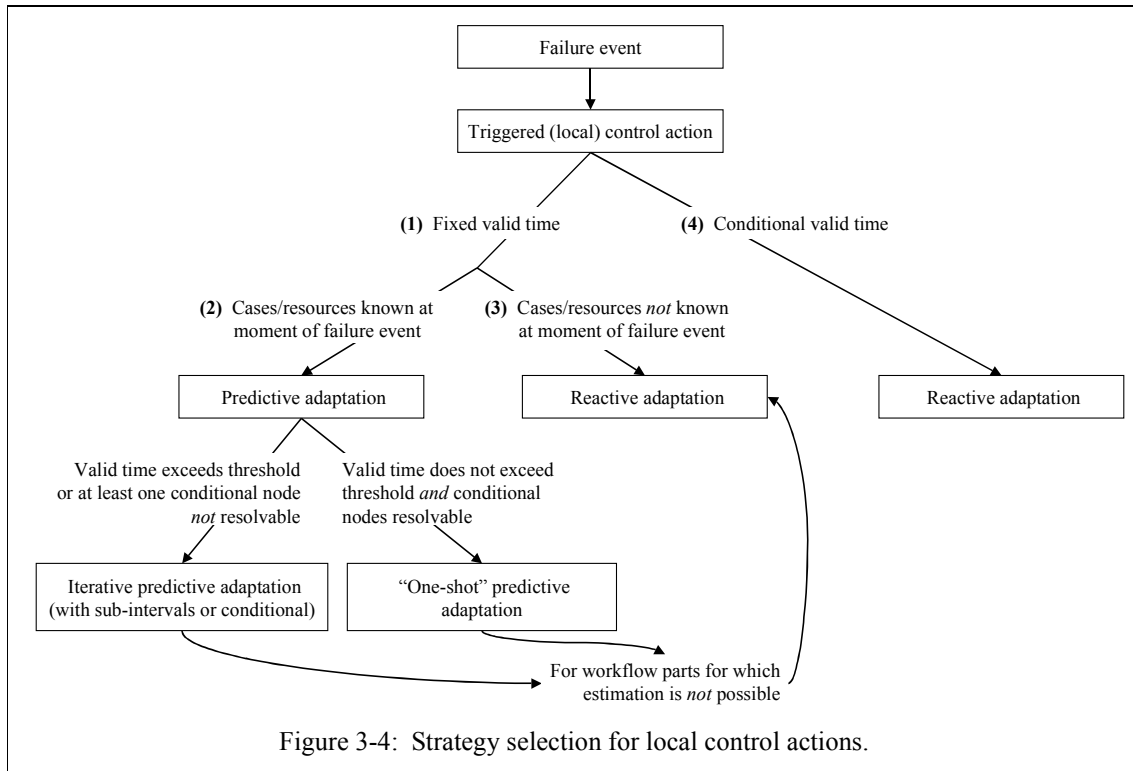
The strategy of predictive adaptation is selected if the following two conditions hold:

- Condition 1: A *fixed* valid time *VT* is assigned to the local control action, i.e., it is known at the moment of the failure event for how long the control action will hold (see **(1)** in Figure 3-4).
- Condition 2: At the moment of the failure event, the cases respective resources assigned to the affected activities are already known (see **(2)** in Figure 3-4). For example, for a

$drop(B, C)$ control action with fixed valid time VT and with B being the activity pattern defined in (viii), predictive adaptation can be used for those nodes administering ETOPOSID for which it is known at the failure moment that they will be executed for case C (during VT). A typical workflow where it is already known at the moment of the failure event for which cases the activity nodes will be executed, is the chemotherapy workflow in Figure 1-1. For this workflow, *all* activity nodes are executed for *one* case which is assigned to the workflow at initialization time.

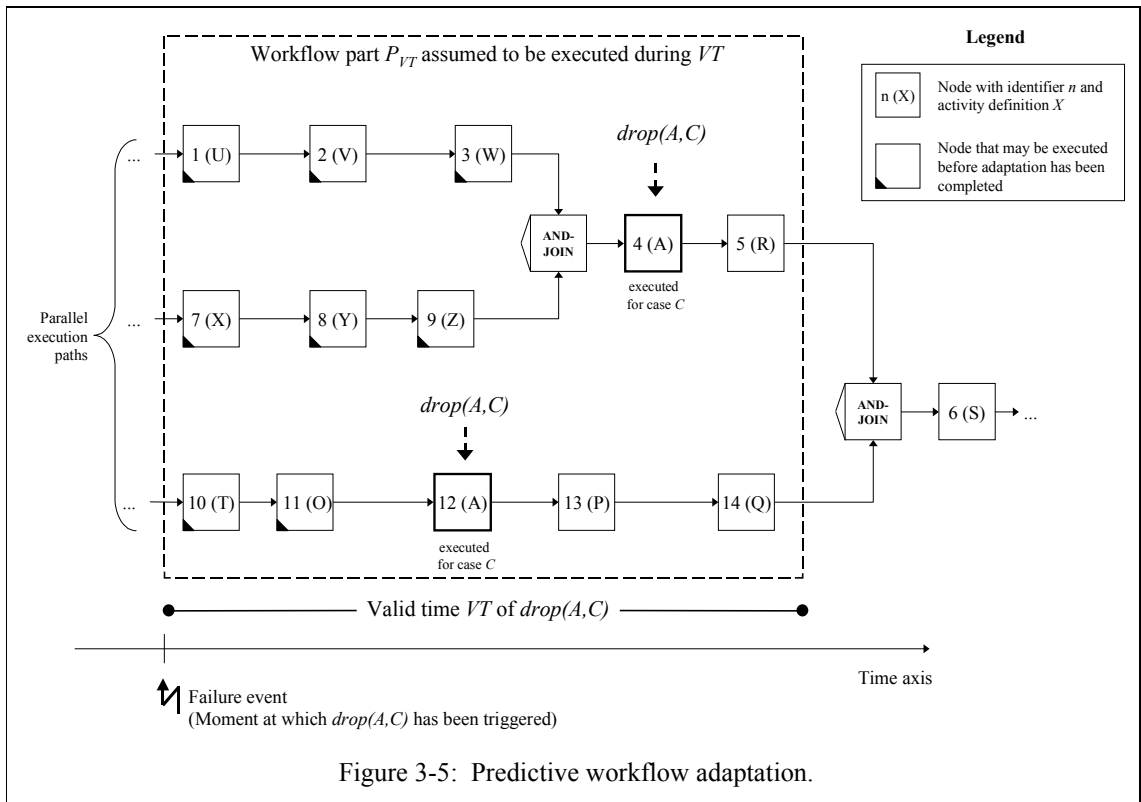
Condition 1 for predictive adaptation means that the temporal interval VT during which the control action is valid is exactly known already at the moment of the failure event. Therefore AGENTWORK can estimate which workflow part P_{VT} will be executed during this assigned fixed VT . This is done by using temporal meta information about the duration of workflow activities (see 3.4.2 below).

Condition 2 means that for the workflow part P_{VT} that is assumed to be executed during VT it is known which activity nodes are affected by the control action as the cases and resources assigned to the activity nodes are already known. Thus, for P_{VT} operations translating the control action into structural workflow adaptations can be performed. For example, in Figure 3-5, first the workflow part P_{VT} that will be executed during the fixed valid time VT of the case-related $drop(A, C)$ control



action is estimated. Second, if we assume that it is known at the moment of the failure event that the two A -nodes 4 and 12 have to be executed for case C , both nodes can be dropped predictively from the control flow. Furthermore, the data flow may have to adapted as well after the control adaptation. For example, if a node n of the remaining control flow in Figure 3-5 needs output data from one of the dropped A -nodes, it may be necessary to compensate the dropping of the A -node by generating a new data flow edge for n which retrieves the needed data from external data sources.

An adaptation may require user interaction (e.g., a confirmation) and therefore may have a duration that is not negligible. For example, if A in Figure 3-5 is some drug administration activity, the physician that has to confirm the dropping of this drug may be absent. Therefore, to avoid an execution delay AGENTWORK continues a workflow during such a predictive adaptation as long as possible. For example, in Figure 3-5 the three paths starting at nodes 1, 7 and 10 are continued while the A -nodes 4 and 12 are dropped from the control flow. However, it cannot be assumed in general that the adaptation will be completed when the control flow reaches a node affected by this adaptation. For example, the execution may reach node 12 before it has been dropped (as the system is waiting for the user confirmation to drop this node). In such a situation, AGENTWORK interrupts a path execution if the node to be executed next is affected by a control action, and if the necessary adaptation



operations have *not* yet been performed for this node (e.g., in Figure 3-5 the path starting at node 10 would have to interrupted after node 11 if node 12 has not yet been dropped). After the adaptation operations have been performed, the path execution can be continued.

If the fixed valid time is very long as its duration exceeds some specified upper bound threshold, AGENTWORK selects an **iterative predictive adaptation**. This means, that instead of estimating the whole part corresponding to the valid time at once (“one-shot” **predictive adaptation**), this valid time is divided into several sub-intervals VT_1, VT_2, \dots, VT_n (where the duration of each sub-interval does not exceed the threshold). Then, it is estimated for VT_1 which workflow part P_1 will be executed during VT_1 , and the adaptation is only applied to P_1 . After P_1 has been executed, the procedure is continued for VT_2 and so on. The question *which* valid time duration threshold should be selected to decide between “one-shot” and iterative predictive adaptation is an empirical matter as it depends on the application domain and the quality of workflow estimation. AGENTWORK allows to specify this threshold as a configuration parameter at installation time and to refine it during the operational work of the system.

An alternative to such an **iterative predictive adaptation with sub-intervals** is the so-called **conditional iterative predictive adaptation**. This means that AGENTWORK at most estimates the workflow only until the next conditional control flow node, i.e., until the next OR-SPLIT or LOOP-END node¹. For this workflow part, the workflow is adapted according to the valid control actions, and continued. When workflow execution reaches the next OR-SPLIT or LOOP-END node, AGENTWORK again estimates the remaining workflow part until the next conditional control flow node, adapts this part, continues the workflow and so on. This is iteratively done until the valid time interval elapses. This strategy is typically selected when there is no possibility to resolve an OR-SPLIT or LOOP-END node predictively (see Section 3.4.2).

3.4.1.2 Reactive Adaptation

The strategy of reactive adaptation is selected whenever the conditions for predictive adaptation are not met (see (3) and (4) in Figure 3-4). For example, if a *conditional* valid time is assigned to a control action, it is not possible to predict for how long the control action will hold (violation of condition 1 for predictive adaptation). Consequently, it is also not possible to derive which part of the remaining workflow will be executed during the valid time of the control action. For example, for a $drop(A, C)$ control action with conditional valid time it is unknown whether an A -node of the remaining control flow will be executed during this valid time or not. A reactive adaptation is also selected even if a fixed valid time is assigned to the control action, but if it is unknown at the moment of the failure event which cases respective resources will be assigned to the affected activity nodes of the remaining control flow (violation of condition 2 for predictive adaptation). For example, in case of a $drop(A, C)$ control action it may not necessarily be known at the moment of the failure whether an A -node of the remaining control flow will be executed for case C or for some other case C' . A typical sample workflow for such a situation may be found at a radiological ambu-

1. In AGENTWORK, a loop termination condition is specified at the LOOP-END node as loops have a REPEAT-UNTIL semantics.

lance. Such a workflow could contain a loop where the loop body consists of radiological examinations and where every loop iteration usually deals with a different patient. Thus, even if a workflow estimation may be possible, it is unknown at the moment of the failure event whether affected nodes in the loop sequence will be executed for the case referenced in the control action. The reactive strategy is also selected, if the conditions for the predictive adaptation hold but if a workflow estimation is *not* possible. For example, this may be because of unresolvable OR-SPLIT nodes or insufficient temporal information about the workflow activities.

Reactive adaptation handles a node directly before it shall be executed. For example, if a *drop(A,C)* control action has been triggered, it is checked for every node *n* that is reached by the control flow during the valid time interval assigned to *drop(A,C)*, whether *n* is an *A*-node that shall be executed for case *C*. If this is the case, the *A*-node is dropped from the control flow. Analogously to predictive adaptation, data flow edges may also have to be removed, adjusted or inserted. After the node has been handled according to the triggered control action, the affected execution path is continued. This procedure is repeated until the valid time of the control action expires.

3.4.2 Workflow Estimation

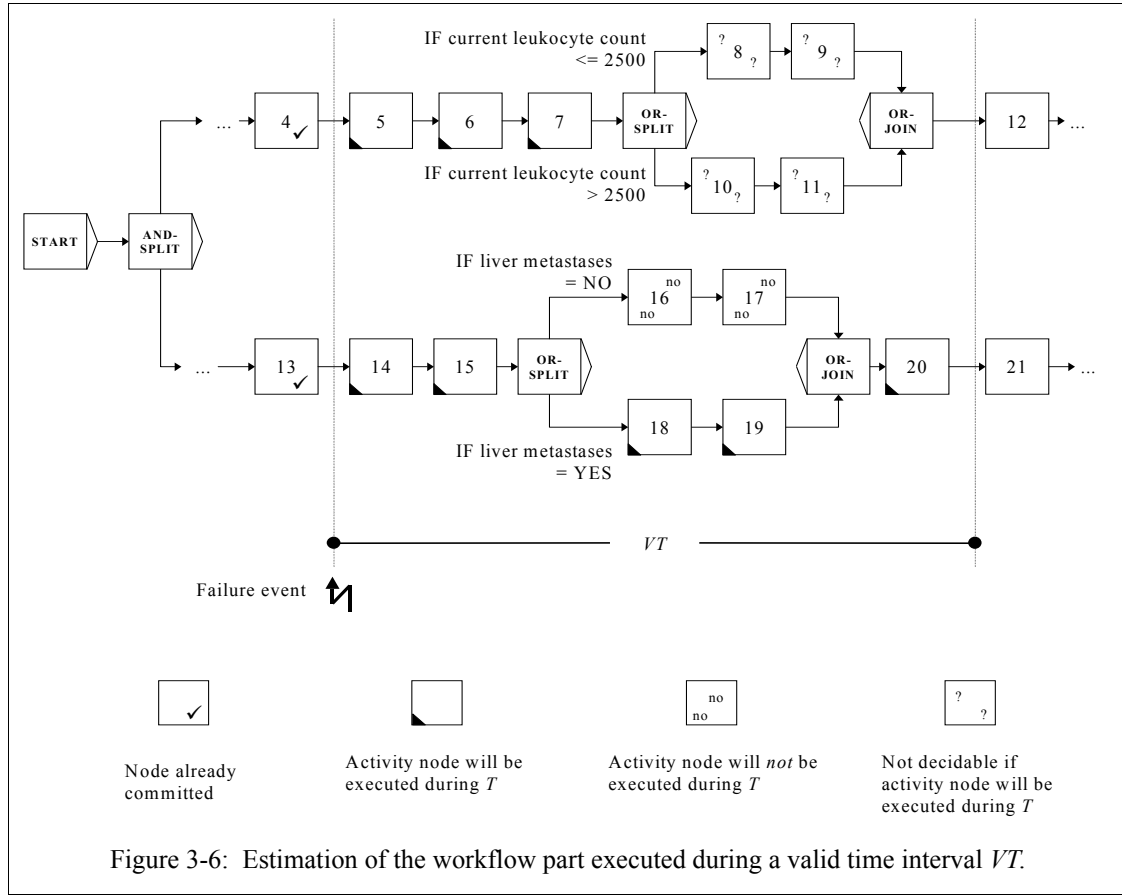
We now sketch the principles of workflow estimation (Figure 3-6). In the following, *VT* denotes the valid time assigned to a control action. For the purpose of this overview, we simply assume that *VT* starts at the moment when the control action has been triggered, and that *VT* does not consist of several *unconnected* parts (i.e., we do not allow that the valid time has a structure such as “*during the next two days and during the 4th and 5th day from now on, but not during the 3rd day*”). Workflow estimation then tries to estimate that part of the remaining workflow that will be executed during *VT*.

Principally, AGENTWORK performs an average-based workflow estimation. This means that for the estimation the *expected average duration* of activities is used. AGENTWORK supports two principal possibilities to obtain such average activity duration information:

- The workflow modeler can assign an expected average duration to every activity definition.
- The workflow engine measures the duration for every activity execution and calculates the average duration of activities based on a given activity definition.

A useful combination of these two possibilities could be to use duration information specified at workflow definition time for the first phase of an AGENTWORK installation and then to continuously improve this duration information by temporal measurements done at workflow execution time.

Workflow parts are then estimated as follows: For each node which has been currently executed or prepared for execution in the moment of the failure event, the control flow path starting from this node is explored. For each path, the average activity durations of the activity nodes are iteratively summed up to estimate how long the execution of such a path will take. The execution of control nodes such as an AND-SPLIT and control flow edges is assumed to have – compared with the



duration of activity durations – a *negligible* duration, unless not specified otherwise in the workflow definition. Duration information w.r.t. data accesses is also obtained from the workflow engine which measures the average duration of data accesses ordered by the different data sources. The problem that an activity execution or a data access may last unexpectedly long – for example because of system failures – cannot be considered the workflow estimation. Rather, such an unexpected delay of workflow execution would be detected by during the workflow monitoring and may induce a correction of the estimation and a readaptation of the workflow (see Section 3.4.5).

When an **AND-SPLIT** is discovered during the exploration of such a path, AGENTWORK continues with the procedure for each of the parallel paths. If it discovers an **OR-SPLIT**, it tries to decide which of the paths starting at the OR-SPLIT will qualify for execution. This may be possible, for example, if the nodes executed at the moment of the failure event are “close” to the OR-SPLIT. Then, data needed for determining which paths will qualify for execution may already be available. For example, in Figure 3-6 such an OR-SPLIT is detected when exploring the path consisting of the nodes 14 and 15. One path qualifies for execution if the cancer patient treated by the workflow has

liver metastases, the other one if the patient has *no* liver metastases. Let us assume for this example, that at the moment of the failure event it is already definitely known that the patient has liver metastases². Therefore, it can be assumed that the path 14→15→18→19→20 will be executed and not 14→15→16→17→20. However, for the other OR-SPLIT after node 7, the branching condition is that the current leukocyte count is less than 2500 respective equal or higher than 2500. Usually, the medical definition of “current” w.r.t. a leukocyte count is that the value has to be measured at the current day (i.e., the day at which the OR-SPLIT is actually executed). Thus, it cannot be determined at adaptation time whether the path starting at node 8 or at node 10 will be executed if it has estimated that the execution of path 5→6→7 will take more than one day. Thus, for the nodes 8, 9, 10 and 11, the failure handling process has to shift to the reactive adaptation (as already described in Section 3.4.1). The problem that data that have been used to predict the OR-SPLIT behavior may unexpectedly change (e.g., the patient suddenly may not have liver metastases anymore) is handled by workflow monitoring after the adaptation. For example, if it would be detected that the path 14→15→16→17→20 in Figure 3-6 would be executed unexpectedly (instead of 14→15→18→19→20), AGENTWORK would re-estimate and re-adapt the workflow if necessary (see Section 3.4.5).

All details and further aspects such as the estimation of loops will be described in Chapter 6 (*Workflow Duration Estimation*).

3.4.3 Structural Workflow Adaptation

Independently from whether reactive or predictive adaptation has been selected, a triggered local control action has to be translated into structural adaptations of the affected control and data flow. For this, AGENTWORK provides a set of *adaptation operators*. The difference between control actions on one side and such adaptations operators on the other side is that control actions express how to cope with a failure event from the point of view of the user on a high level of abstraction. In particular, control actions do not take care about syntactical and structural details of workflow definitions. In contrast to this, adaptation operators deal with the adaptation on the syntactical and structural level (e.g., they remove, insert or replace nodes and edges to satisfy the control actions).

AGENTWORK distinguishes between *control flow operators* that adapt the control flow according to triggered control actions, and *data flow operators* that adjust the data flow if necessary. As the data flow operators can only become clear after the data flow model of AGENTWORK has been described in Chapter 5 (*Workflow Definition and Execution*), we give only two examples for control flow operators (Figure 3-7) in this overview chapter. The operator

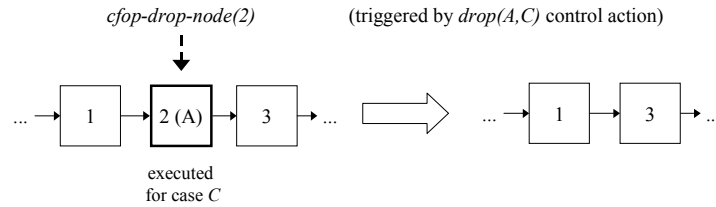
*cfop-drop-node(node-id)*³

is used if a *drop* control action has been triggered (Section 3.3.1). This operator takes the identifier of a node within a sequence as input and removes it from this sequence (Figure 3-7 a). The operator

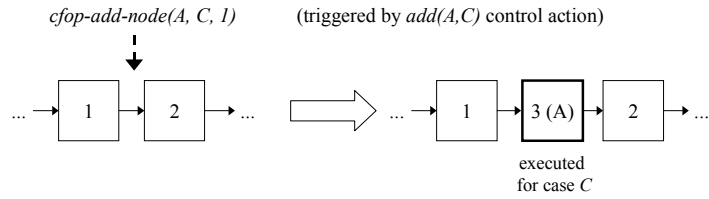
2. As metastases usually do not occur “over night”, this assumption makes sense.
3. *cfop* stands for “control flow operator”.

Figure 3-7: Examples for control flow adaptation operators.

a) $cfop\text{-}drop\text{-}node(node\text{-}id)$



b) $cfop\text{-}add\text{-}node(activity\text{-}def, case\text{-}id, predecessor\text{-}id)$

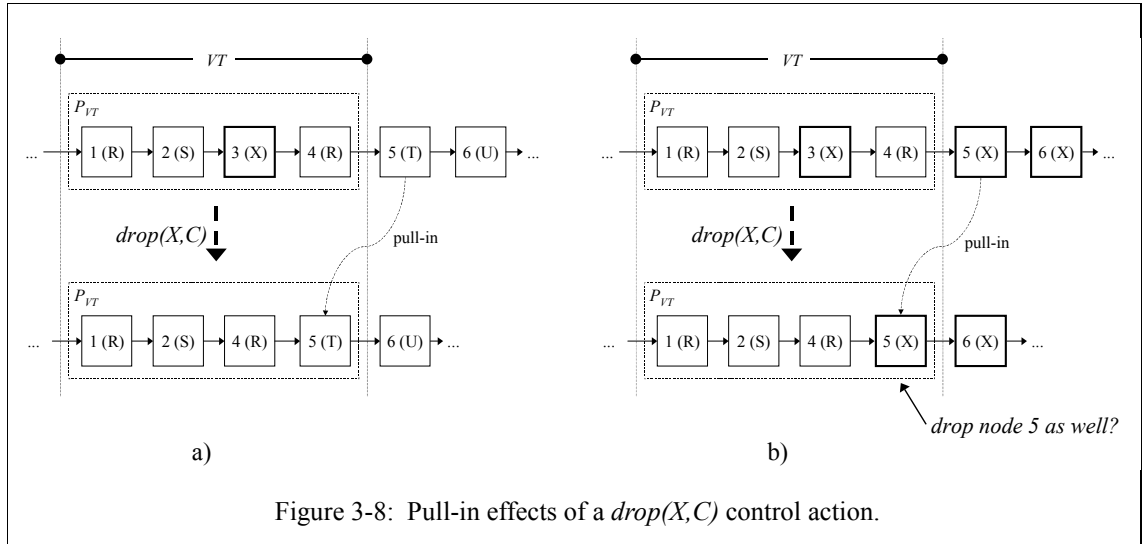


$cfop\text{-}add\text{-}node(activity\text{-}def, case\text{-}obj, node\text{-}id)$

is used if an *add* control action (Section 3.3.1) has been triggered. This operator takes as input the activity definition that shall be assigned to the new node (*activity-def*), the case object for which the new node shall be executed, and the identifier of the activity node behind which the new node shall be added (*node-id*). The operator then generates a new node based on this activity definition and inserts it after the specified node (Figure 3-7 b). AGENTWORK provides several variations of these operators which are used depending on whether the affected nodes belong to parallel or conditional execution parts, or to loops. The full set of control flow operators will be described in Chapter 8 (*Structural Adaptation Operators*). There it will be also described how it is decided *where* a new node has to be inserted in case of an *add* control action.

3.4.4 Adaptation Side-Effects

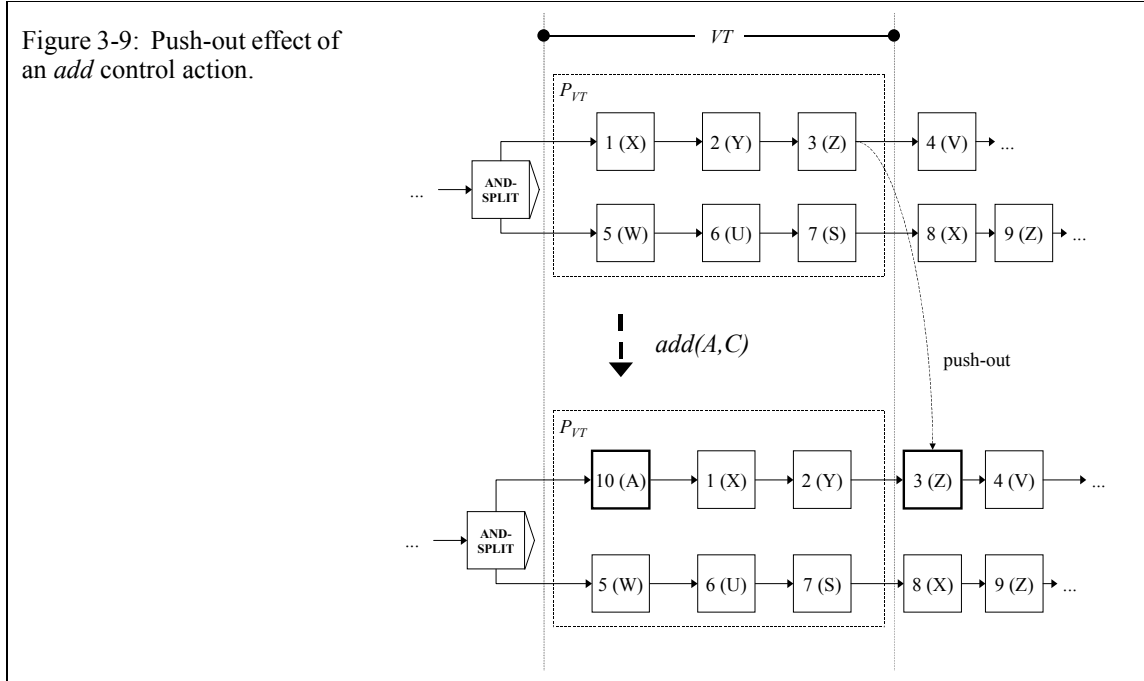
The application of structural adaptation operators such as those shown in Figure 3-7 may have some side effects that can lead to considerable adaptation “anomalies”. We distinguish so-called *pull-in* and *push-out* effects. As these effects and their handling has influenced the design of AGENTWORK significantly, we discuss them already in this overview chapter. With P_{VT} we again denote that part of a workflow that is (assumed to be) executed during a valid time interval VT . Furthermore, we assume that the activity nodes of the workflows used in the examples of this section are always executed for the same case C .



3.4.4.1 Pull-in Effects

Pull-in effects denote effects where the translation of a control action into a structural workflow adaptation causes that nodes so far not belonging to P_{VT} become a member of P_{VT} (i.e., they are “pulled into” P_{VT}). For example, in Figure 3-8 a) P_{VT} initially consists of the nodes 1, 2, 3 and 4. Node 3 with activity definition X is removed because of a $drop(X,C)$ control action. As a consequence, node 5 now will be executed during VT so that node 5 becomes a member of P_{VT} . At first glance, this mechanism of dynamically adjusting P_{VT} looks appropriate as nodes such as the T -node 5 now can be executed earlier which usually will be desirable. However, there is at least one problem which is illustrated in Figure 3-8 b). There, node 5 also would have to be pulled into P_{VT} analogously to the situation in Figure 3-8 a). However, in Figure 3-8 b) node 5 is a X -node as well. Thus, after having been pulled into P_{VT} it also would have to be dropped as the semantics of $drop(X,C)$ with valid time VT is that *no* X -node must be executed during VT (see 3.3.4). The same would then happen to X -node 6. If such a X -node for example would represent a drug administration, this would mean that not only drug administrations within P_{VT} would be dropped but also some that originally were scheduled for execution beyond VT . As an even more extreme example, imagine that the workflow in Figure 3-8 b) would exclusively consist of X -nodes from node 5 on (e.g., of a sequence of 10 X -nodes). If we would allow an uncontrolled pull-in of these X -nodes into P_{VT} , all these 10 X -nodes would be removed.

The question is whether this really is intended by control actions such as $drop(X,C)$ with valid time VT . At least, those pull-in effects should be carefully controlled or even avoided where nodes that originally would have been executed *beyond* VT would be affected by a control action valid *during* VT , if pulled into P_{VT} . In particular, the user should be requested whether a pull-in should be allowed or not.

Figure 3-9: Push-out effect of an *add* control action.

3.4.4.2 Push-Out Effects

Vice versa, an adaptation may also “push out” some nodes from P_{VT} . For example, in Figure 3-9 an *A*-node (node 10) is inserted for case *C* into the sequence $1 \rightarrow 2 \rightarrow 3$ because of an $add(A, C)$ control action. Because of this additional node, the *Z*-node 3 will not be executed anymore during VT and therefore would not belong to P_{VT} anymore, i.e., it is “pushed out” from P_{VT} . Generally, this may be acceptable. However, there is at least one problem if other control actions hold during VT as well. For example, imagine that in Figure 3-9 a $drop(Z, C)$ control action holds during VT as well. Then, the push-out effect of $add(A, C)$ would mean that *Z*-node 3 would not be “reached” anymore by the $drop(Z, C)$ control action. At first glance, this may also be acceptable as the constraint expressed by $drop(Z, C)$ (i.e., that no *Z*-node may be executed during VT) is met. However, in this case this is met not by removing the *Z*-node but by pushing it out from P_{VT} . In particular, this is a problem of the *order* of control action processing. If the $drop(Z, C)$ control action is processed first and the $add(A, C)$ second, then *Z*-node 3 is removed from P_{VT} and not pushed out. If the opposite order is used, then the *Z*-node 3 is pushed out from P_{VT} but *not* removed. Thus, analogously to pull-in effects, one should carefully control or even avoid those push-out effects where nodes are pushed out from P_{VT} by a control action though they are affected by other control actions being valid during VT . In particular, the user should be requested whether a push-out should be allowed or avoided.

Pull-in and push-out effects are not only a problem of predictive adaptation, as Figure 3-8 and Figure 3-9 may suggest. They can also occur in the context of reactive adaptation, and additionally show another weakness of this adaptation strategy: As reactive adaptation does not estimate a workflow and thus has no notion of P_{VT} , this strategy even would *not notice* subsequent pull-in or push-out effects. For example, reactive adaptation would simply perform the subsequent pull-in and dropping of X -nodes in Figure 3-8 b) without any possibility to intervene, as reactive adaptation does not “know” that the X -nodes pulled in originally did not belong to the workflow part corresponding to the valid time VT .

One may argue that the problem of push-out and pull-effects arises only because of the particular structure of control flow failure rules as suggested in Section 3.3. In particular, the fact that a control action such as $drop(A, C)$ only states that no A -nodes may be executed *during some valid time* VT causes this problem. Thus, one may suggest alternative rules that consist of a tighter coupling between events and local workflow structures such as activity nodes, e.g.,

(xii)

WHEN new hematological finding of patient P
 WITH leukocyte count $< 1000 \text{ \#}/\text{mm}^3$
 THEN drop node n (for workflows based on W and running for P)

where n is the identifier of a node applying the drug *Etoposid*. Then, one would not have to deal with pull-in or push-out effects as node n is removed anyway, regardless how it is “pulled” or “pushed” by other control actions. However, this approach is not appropriate because of the following reasons:

- First, such tightly-coupled rules as in (xii) do not provide the level of abstraction that has been considered necessary according to requirement 1 (*Representation of Failure Events and Control Actions*) discussed in Section 1.3. In particular, there it has been identified that rules for control flow failures should express the point of view of the workflow user (e.g., the physician) without making any assumptions about the structure of the workflows. In contrast to this, rules such as (xii) require that it is known at rule definition time in which workflows affected nodes are located and what the value of their identifier is.
- Second, when the workflow modeler rearranges workflow definitions by moving activity nodes from one workflow definition to another, tightly-coupled local failure rules such as (xii) would also have to be rearranged as they directly reference workflow structures. This complicates the maintenance of a rule base consisting of rules for control flow failures.

The handling of pull-in and push-out effects will be described in detail in Chapter 8 (*Structural Adaptation Operators*) and Chapter 9 (*Predictive Control Flow Adaptation*).

3.4.5 Workflow Monitoring

Workflow monitoring is necessary in the context of predictive adaptation to check whether the temporal estimation on which an adaptation is based matches the actual execution of the adapted work-

flow. For example, subsequent adaptations adding or dropping nodes, or technical errors may have the consequence that estimations become invalid. We distinguish two principal mismatch types between estimations on one side and the actual execution on the other side, namely *temporal acceleration* and *temporal delay* (Figure 3-10). Let P_{VT} again denote the workflow part that is assumed to be executed during a valid time VT , and let $exec-time(P_{VT})$ denote the interval actually needed to execute P_{VT} .

3.4.5.1 Temporal Acceleration

P_{VT} is executed *faster* than it has been estimated. This means that workflow parts which have not been considered so far are now as well going to be executed during VT . For example, due to a faster execution of the nodes 1-8 in Figure 3-10 b), an additional workflow part consisting of nodes 9-11 in Figure 3-10 b) may now be executed during VT as well. This part then may have to be adapted as well to satisfy control actions that are valid during VT . In particular, it has to be considered that such an adaptation due to a temporal acceleration may lead to pull-in or push-out anomalies as described in Section 3.4.4.

3.4.5.2 Temporal Delay

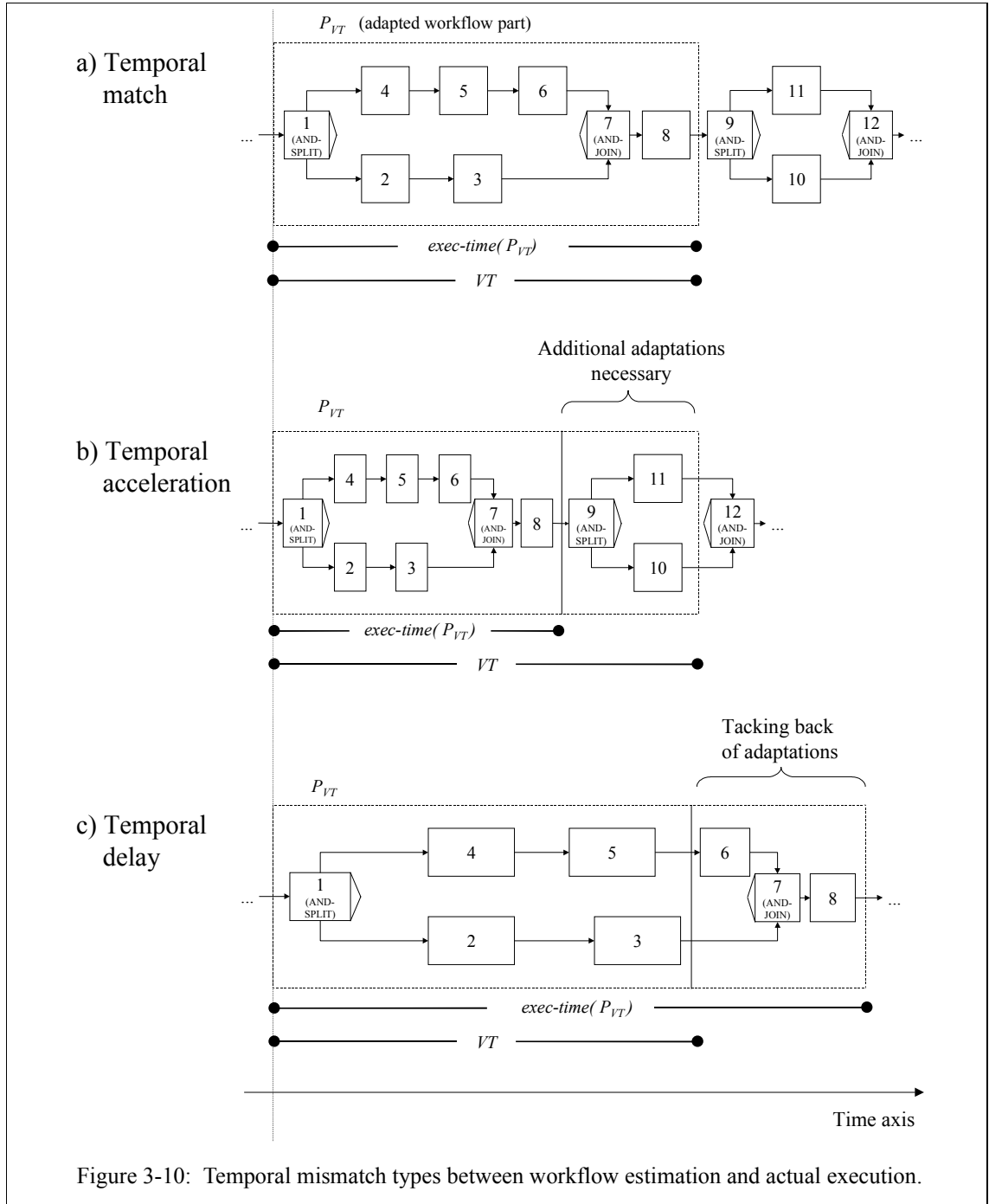
The execution of P_{VT} is delayed. This means that parts of P_{VT} that have been assumed to be executed during VT will not be executed anymore during VT . For example, in Figure 3-10 c) the execution of the nodes 1-5 has taken unexpectedly long. Therefore, adaptations that have been performed for the part that will not be executed anymore during VT (e.g., the part consisting of nodes 6-8 in Figure 3-10 c) may have to be taken back again. Analogously to temporal acceleration, pull-in or push-out anomalies have to be considered.

Details about the workflow monitoring approach of AGENTWORK are described in Chapter 9 (*Predictive Control Flow Adaptation*).

3.5 Summary and Discussion

In this chapter we have given an overview of AGENTWORK. First, we have characterized the principal modeling approach. This approach provides a *global data schema* to relieve the AGENTWORK components from data distribution and heterogeneity, *control actions* that express what has to be done with workflows or activities, and *rules for control flow failures* stating which application events trigger which control actions. Second, it has been sketched under which conditions which adaptation strategy (reactive or predictive) is selected to handle triggered control actions, and how these different strategies are performed. It has further been identified that *temporal* aspects play a substantial role for predictive adaptation. This is because predictive adaptation has to estimate workflow durations, so that the duration of activities, parallel and conditional execution paths has to be determined. Thus, a precise and appropriate notion of “time” and “duration” is required and consequently will be given in the following chapters.

It could be asked why predictive adaptation is supported at all by AGENTWORK, though it seems to



cover only a minority of cases according to Figure 3-4, and though its complexity is very high as workflow estimation and workflow monitoring is required. This is a matter of the percentage of control actions with fixed valid time and of the question for how many workflows the cases for which the activities are executed are known already at workflow initialization time (as then predictive adaptation can be performed). For example, in hemato-oncology most failure handling rules can be defined with a fixed valid time as physicians have quite precise temporal heuristics for how long a drug should be dropped or given additionally. Furthermore, in medical domains it can be observed that many treatment workflows (such as the one in Figure 1-1) are typically designed to be executed for exactly one patient during their life span. This avoids mixing up a treatment workflow with different patients as this would increase the danger of administering the wrong drugs and dosages to the wrong patients. Therefore, the failure handling rules and workflows meeting the conditions for predictive adaptation form a large class that cannot be neglected. Together with the advantages of predictive adaptation listed in Chapter 1, it can be clearly said that for the medical domain the costs for supporting this strategy are justified. It is the assumption of the thesis that this also holds for many other application domains. For example, typical workflows supporting contract or damage processing in insurance business usually are executed for one and only one customer during their life span.

Before we describe the details of handling control flow failures in the chapters 5-11, we now introduce the data and rule definition language of AGENTWORK in Chapter 4 (*Data and Rule Definition with ActiveTFL*). Because of the above-mentioned importance of temporal aspects, this language focuses not only on data and rule modeling but also on the modeling of time and temporal operators.

Data and Rule Definition with ACTIVETFL

In this chapter we introduce ACTIVETFL (*Active Temporal Frame Logic*) which is the AGENTWORK data and rule definition language. ACTIVETFL will be used in the following to specify data, events, control actions and failure rules. Basically, ACTIVETFL combines a temporal logic and active rule elements known from the field of active databases. ACTIVETFL is *not* a general-purpose active rule language. Rather, it has been designed for the specific purposes of this thesis.

The chapter is organized as follows: In Section 4.1 we identify the requirements that should be met by a data and rule definition language in the AGENTWORK context, and show that a *temporal logic* meets central requirements better than other language types. This is mainly because important phases of handling control flow failures can be viewed as a deductive process over temporal structures. Section 4.2 describes *Frame Logic*, an object-oriented logic which serves as the basis of ACTIVETFL. In Section 4.3, we extend Frame Logic by several elements known from temporal logics to achieve better *temporal* support. In Section 4.4, we describe the notion of *active* rules in ACTIVETFL. The chapter concludes with a summary in 4.5.

4.1 Language Requirements

For data and rule definition purposes a broad range of specification languages exists, such as SQL¹-based languages (e.g., SQL92 [DATE & DARWEN 1997], SQL:99 [EISENBERG & MELTON 1999]), ODL² [CATTELL ET AL. 2000], XML schema languages³ [MARCHAL 1999], or logical languages

1. Structured Query Language

(e.g., Description Logic [BRACHMAN 1977, CALVANESE ET AL. 1998], Frame Logic [KIFER ET AL. 1995]). They significantly differ w.r.t. the supported data model, the support of rules and temporal aspects, and the way the semantics is formalized. To identify the most appropriate language type, we list the requirements that should be met by an AGENTWORK data and rule definition language and discuss how the different languages meet these requirements.

4.1.1 Support of Object-Oriented or Object-Relational Data Models

It has been widely recognized that in many application domains object-oriented or object-relational modeling capabilities are required to cope with data complexity. This holds especially for the medical domain. For example, in former studies performed by the author it is shown that in hemato-oncology the broad range of different medical findings and activities can best be modeled by specialization hierarchies and aggregation, and that the numerous causal and temporal associations between medical data entries are best supported by an object-oriented data model [MÜLLER 1997, MÜLLER 1994, MÜLLER ET AL. 1997 B]. In particular, the object-oriented or object-relational data model allows to define application data on a high-level of abstraction understandable also for end users (e.g., workflow users).

Object-oriented or object-relational data models are supported, for instance, by SQL:99, ODL, and several logical languages (e.g., Frame Logic). In contrast to this, languages based on the relational model [CODD 1970] (such as SQL92) provide limited support concerning specialization, association and aggregation. The same holds for “classical” first-order predicate logic as it closely corresponds to the relational model [REITER 1984].

4.1.2 Support of Rule Definitions

As sketched in Chapter 3, control flow failures can be handled in an intuitive manner by event-condition-action rules (ECA rules). Such a rule specifies in its event-condition part under which condition an event may raise a control flow failure. The action part states which control action has to be performed to cope with the control flow failure. Therefore, the support of *rules* must be viewed as a mandatory requirement for an AGENTWORK data definition language.

Rules are supported by SQL:99 and logical languages. In contrast to this, ODL and XML do not provide any explicit constructs for rules. Recent efforts that have been undertaken to add rule-oriented elements to ODL [SAMPAIO & PATON 2000] have not been considered for this thesis as the results provided are of only preliminary nature so far.

4.1.3 Support of Temporal Aspects

As already discussed in Chapter 1 and Chapter 3, *temporal* aspects of events and actions cannot be neglected in the domains addressed by AGENTWORK. In particular, the following temporal support

-
2. Object Definition Language
 3. eXtensible Markup Language

has to be viewed as mandatory (we do not discuss ODL and XML in this section as they do not provide any temporal support).

4.1.3.1 *Fixed Valid Time*

Action parts such as (see Section 3.3.4)

THEN drop(B, P) during the next seven days

require that valid time intervals which are specified by a fixed number of points in time can be assigned to actions. This is supported by temporal extensions of SQL such as TSQL2 [SNODGRASS ET AL. 1995] and SQL:99/TEMPORAL which is currently proposed as a temporal extension for the SQL:99 standard [SNODGRASS ET AL. 1998, SNODGRASS 1999]. Furthermore, several logical languages have been extended with a fixed valid time dimension [BAUDINET ET AL. 1993, CHOMICKI & TOMAN 1998].

4.1.3.2 *Conditional Valid Time*

Action parts such as (see Section 3.3.4)

THEN drop(B, P) until leukocyte count of P is higher than 2500 again (i)

require that a valid time can be specified conditionally by a termination condition on data.

A dependency such as (i) is difficult to represent with current temporal SQL extensions, as *fixed* valid time intervals cannot be used for this. For example, to specify that the drug ETOPOSID has to be dropped when the leukocyte count is less than 1000 and that it can be administered again when the leukocyte count becomes higher than 2500, in SQL:99 a trigger such as the *drop-or-redrop-etoposid* trigger in Table 4-1 would have to be defined. This trigger reacts when the leukocyte count is less than 1000 or when the leukocyte count becomes higher than 2500. In the first case, it inserts a drop instruction into a table *drug-orders* to express that ETOPOSID must not be administered anymore. In the second case, it deletes the ETOPOSID drop instruction again. However, this solution has two disadvantages:

- First, for a domain such as hemato-oncology several hundreds of dependencies of type (i) exist. Therefore, representing each dependency by triggers with such large conditional action parts as in (i) would reduce readability.
- Second, and more important, trigger definitions such as those shown in Table 4-1 may have negative side-effects. For example, let us assume that there is another trigger dropping the drug ETOPOSID because of critical *renal* values by also inserting a drop instruction into the table *drug-orders*. Then, the trigger *drop-or-redrop-etoposid* would also delete this drop instruction when the leukocyte count becomes higher than 2500 though the renal values still may be critical. Certainly one could add additional columns (such as the reason for dropping the drug) into the *drug-orders* table. Then, the second case statement of *drop-or-redrop-etoposid* would additionally have to check whether the reason for the dropping really was a hematological one.

Notation in SQL:99	Description
<pre>CREATE TRIGGER drop-or-redrop-etoposid AFTER INSERT ON laboratory-findings REFERENCING NEW AS new WHEN new.parameter = "Leukocyte-Count" AND (new.value < 1000 OR new.value > 2500) BEGIN ATOMIC CASE WHEN new.value < 1000 THEN INSERT INTO drug-orders values (new.pat-id, "ETOPOSID", drop) CASE WHEN new.value > 2500 THEN DELETE FROM drug-orders WHERE pid = new.pat-id AND name = "ETOPOSID" AND order-type = drop END</pre>	<p>When a tuple is inserted into table <i>laboratory-findings</i> stating that a patient has a leukocyte count less than 1000 or higher than 2500, then do the following:</p> <p>If new leukocyte count less than 1000: Insert tuple in table <i>drug-orders</i> stating that respective patient must not get ETOPOSID anymore.</p> <p>If new leukocyte count higher than 2500: Delete the tuple in table <i>drug-orders</i> stating that the patient must not get ETOPOSID anymore.</p>

Table 4-1: Two SQL:99 triggers defining a failure rule with conditional valid time.

We assume a table *laboratory-findings*(*lab-id* INTEGER, *pat-id* INTEGER, *parameter* VARCHAR(40), *value* FLOAT) storing laboratory values (omitting any units) and a table *drug-orders*(*pat-id* INTEGER, *name* VARCHAR(40), *order-type* ENUM{add, drop}) storing administration orders for drugs. For both tables, *pat-id* denotes the patient identifier. The trigger syntax is taken from [KULKARNI ET AL. 1999].

However, this makes the triggers more complex and reduces readability additionally, as additional conditions have to be checked.

In contrast to this, temporal logics provide operators to express a broad range of temporal dependencies in a compact manner [CHOMICKI & TOMAN 1998, MANNA & PNUELI 1992]. For example, to represent rules with an action part such as (i) temporal logics provide the so-called *Unless* operator. Informally, a formula such as $p \text{ Unless } q$ states that p holds at least unless q holds or holds forever in case that q never holds. After having introduced such temporal operators formally in 4.3, we will show that dependencies such as (i) can be expressed in temporal logics without the above-mentioned disadvantages. At the moment, the SQL:99 committee does not plan to support temporal operators such as *Unless* in the future⁴, so that SQL:99 does not provide the needed temporal support in the next future.

4.1.3.3 Composite Temporal Events

As already identified as requirement 1 in Chapter 1, an event often will constitute a control flow failure only together with other events. Thus, event-condition parts such as

4. Personal e-mail note received from R. Snodgrass who closely works with the ISO SQL:99 committee on the temporal support of this language.

Language Requirements		SQL92	SQL:99	ODL	XML	Logic Languages
Support of Object-Oriented or Object-Relational Data Models		No	Yes	Yes	Partially (aggregation)	Some (e.g., F-Logic)
Support of Rule Definitions		No	Yes	No	No	Yes
Support of Temporal Aspects	Fixed Valid Time	Yes (TSQL2)	Yes (SQL99/TEMPORAL)	No	No	Some (Temporal Logics)
	Conditional Valid Time	No	Limited (only through triggers)	No	No	Some (Temporal Logics)
	Composite Temporal Events	No	No	No	No	Limited (no time series)
Declarative Semantics		Yes	Yes (relational part)	No	No	Yes

Table 4-2: Language support w.r.t. requirements for handling control flow failures.

WHEN TIME-SERIES(*EC*, 3, “one week”) for *P*

(ii)

(with *EC* being some critical leukocyte count) require to define *composite events*. This is not supported by most of the above-mentioned languages. For example, a SQL:99 trigger can only monitor one event, but not a conjunction or disjunction of events. Temporal logics can combine events by conjunction or disjunction but do not support time series events such as (ii). In contrast to this, several research approaches have suggested event algebras to allow the composition of events [GEHANI ET AL. 1992, CHAKRAVARTHY ET AL. 1994, MOTAKIS & ZANIOLO 1997 A, DINN ET AL. 1999].

4.1.4 Declarative Semantics

As this thesis focuses on the semi-*automated* handling of control flow failures, an important additional aspect is the way the *semantics* of a definition language is defined. It is widely accepted that whenever a significant automation of some process is intended, a *declarative* semantics is of advantage [POOLE ET AL. 1998]. Declarative semantics is available for logical languages and languages based on the relational calculus. It is not available for ODL, XML and for those parts of SQL:99 going beyond the relational calculus (e.g., triggers).

4.1.5 Conclusion

Table 4-2 summarizes the support of the different languages w.r.t. the listed requirements. Because logical languages support most of the requirements, a logic has been selected as the primary data and rule definition language for AGENTWORK. In particular, it had to be decided whether to use Frame Logic [KIFER ET AL. 1995] or Description Logic [BRACHMAN 1977] as core language, as

both logics support the object-oriented data model. Between these two, Frame Logic has been selected because of the following reasons:

- Frame Logic has been designed to specify information systems and database applications. It provides a database-oriented uniform language for classes, objects, functions, methods, predicates, queries and deduction rules. In contrast to this, Description Logic focuses on natural language processing and terminological reasoning [FRANCONI 2002, INGENERF 1993] by strictly separating terminological knowledge (i.e., concepts and their relationships) and assertional knowledge (i.e., facts). However, this focus of Description Logic is not relevant for the purposes of this thesis.
- F-logic is also an extensible logic, as it can be combined with other recently proposed logics for knowledge representation, such as HiLOG [CHEN ET AL. 1993], Transaction Logic [BONNER & KIFER 1994], and Annotated Predicate Logic [KIFER & SUBRAHMANIAN 1992]. Though these extensions are not used in this thesis, they may be useful for future versions of AGENTWORK. For example, Annotated Predicate Logic may be used for an enhanced handling of inconsistency aspects when dealing with control flow failures [KIFER & LOZINSKII 1992].

As Frame Logic in its original version is “non-temporal“, it has been decided to extend it with elements of a reified temporal logic (see 2.5.3) such as fixed and conditional valid time. Furthermore, this temporal Frame Logic is enriched with a composite event model and with active rules. The three layers of the resulting language called ACTIVETFL are illustrated in Figure 4-1.

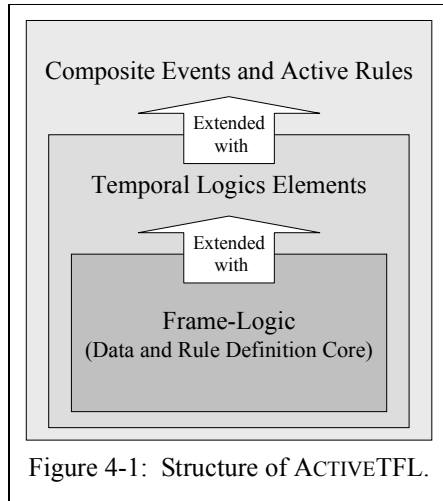


Figure 4-1: Structure of ACTIVETFL.

Taking a logic-based data and rule definition language does *not* imply that relational, object-relational, ODL or XML databases cannot be used as data sources in an AGENTWORK environment. It only means that the AGENTWORK components operate on a global data schema not specified in these languages. However, via the communication and integration layer introduced in Chapter 3, these database types can be accessed.

In the following, we describe the different parts of ACTIVETFL mainly by means of examples.

4.2 Frame Logic (F-Logic)

Frame Logic (F-Logic) [KIFER & LAUSEN 1989, KIFER ET AL. 1995] is an object-oriented logic based on Maier’s O-Logic [MAIER 1986]. Model-theoretic declarative

semantics on the basis of first-order predicate logic and a proof theory are provided. Prototypical implementations of F-Logic have been realized, for example, at the University of Freiburg, Germany [FROHN ET AL. 1997] and the University of Valencia, Spain [CARSI ET AL 1998].

In this thesis the syntax of F-Logic has been partially adapted. This is because some syntactical ele-

ments of F-Logic – such as the “ $=>$ ” and “ \rightarrow ” symbols for set-valued object components – are uncommon for readers not familiar with this logic class. Therefore, such symbols have been replaced by more intuitive notions such as *Set<Type>*. Furthermore, instead of using the classical logic symbols \wedge , \vee , \neg , \in , \exists and \forall , we use *AND*, *OR*, *NOT*, *IN*, *EXISTS* and *FOR-ALL* for a better readability. Nevertheless, the language model and semantics is that of F-Logic as described in [KIFER ET AL. 1995].

We now introduce the main F-Logic elements, namely class definitions, objects (i.e., class instances), object paths, object patterns, methods, object extensions, predicates, formulas, queries, and rules.

4.2.1 *F-Logic Classes and Objects*

4.2.1.1 Class Definitions

F-Logic class definitions are of the form

(iii)

Case[*case-id*: Integer, *name*: String, *activities*: Set<Activity>],

Resource[*name*: String, *needed-to-execute*: Set<Activity>, ...],

Activity[*date*: Date, *time*: Clock-Time, *activity-for*: Case, ...],

Patient IS-A Case, *Physician IS-A Resource*,

Patient[*social-num*: Integer, *diagnosis*: String, *doctor*: Physician],

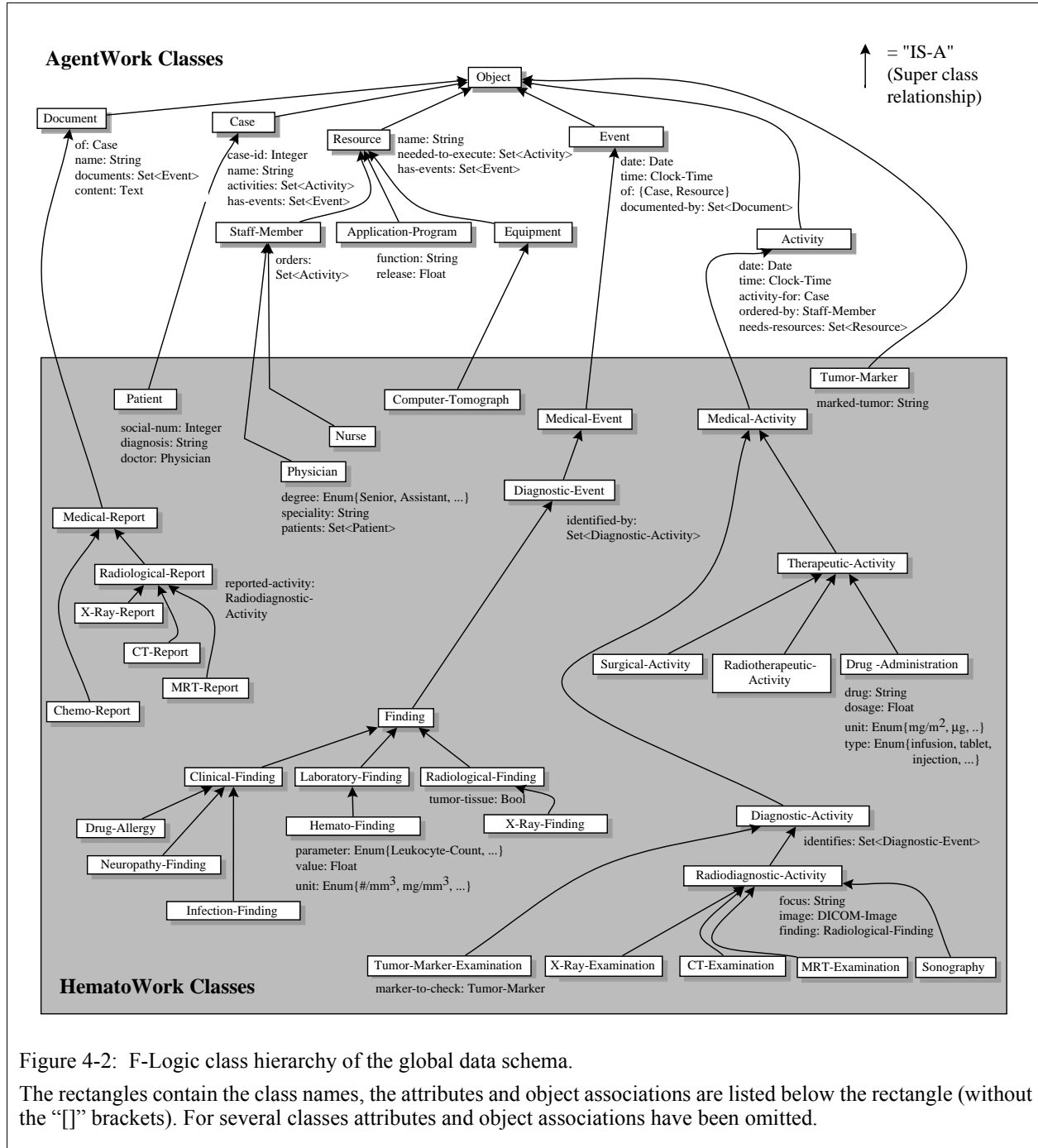
Physician[*degree*: Enum{Senior, Assistant, ..}, *speciality*: String, *patients*: Set<Patient>].

In this example, the classes *Case*, *Resource*, and *Activity* are the F-Logic notations of the UML classes *Case*, *Resource*, and *Activity* in Figure 3-2. For example, the entries

activity-for: *Case* in the class *Activity* and
activities: *Set<Activity>* in the class *Case*

represent the n:1 UML associations between activities and cases of Figure 3-2 (i.e., an activity is executed for exactly one case, and for one case an arbitrary number of activities may be executed). As with SQL:99's REF construct, F-Logic does not specify whether an object association also has an aggregation or composition semantics. This is left up to an class implementation. The symbol "IS-A" denotes the subclass relationship (e.g., *Patient* is a subclass of *Case*). By convention, it always holds $C \text{ IS-A } C$ for an F-Logic class C .

Figure 4-2 shows a graphical notation of the most important F-Logic classes of the global AGENTWORK data schema, including the classes of (iii) and all other classes specified in Figure 3-2. The grey part of Figure 4-2 contains domain-specific subclasses for HEMATOWORK, which we will frequently use in examples of this thesis. As we want to concentrate on the principal handling of control flow failures and not on medical data modeling, we refer to approaches of the author described



in [MÜLLER 1994, MÜLLER 1997]. The HEMATOWORK medical data schema is based on the approaches described in these references.

4.2.1.2 Objects

F-Logic objects (i.e., class instances) are denoted as follows:

bob:Patient [*case-id*: 124576, *name*: “Miller, Bob“, *activities*: {},
social-num: 63728888, *diagnosis*: “Hodgkin-Lymphoma“, *doctor*: *steve*]
steve:Physician [*name*: “Taylor, Steve“, *degree*: Senior, *speciality*: “Oncology“,
patients: {*bob*, *fred*, *mary*}].

with “:” denoting the *is-object-of* relationship (e.g., *bob* is an object of class *Patient*).

The “.”-operator is used to reference components of an object and to navigate along object associations. For example,

bob.doctor.degree

refers to the degree of the doctor treating *bob*.

4.2.1.3 F-Logic Methods and Functions

A class may have methods. In contrast to attributes, methods have round brackets (enclosing optional input parameters) and an optional “:” to indicate the return type. For example, to the class *Physician* we could add the method

patientNames(): Set<String>

returning the names of all patients treated by a physician.

Functions are declared analogously, but are not assigned to classes.

4.2.1.4 Object Paths

An *object path* is any path expression starting from an F-logic object and navigating along the associations of this object. For example, if *d* is a *Drug-Administration* object according to Figure 4-2, then *d.ordered-by.name* is the object path leading to the name of the *Staff-Member* object that ordered *d*. The attributes and methods of an object *o* are viewed as special object paths, i.e., as paths with length 1.

4.2.1.5 Object Patterns

An *object pattern* specifies constraints w.r.t. the internal structure of an object. It has the form

Class[*path-constraints*]

with *Class* being an F-Logic class and *path-constraints* being a set of constraints on paths of *Class* objects. For example, the object definition

$$\text{Hemato-Finding}[\text{parameter} = \text{Leukocyte-Count}, \text{unit} = \text{\#/mm}^3] \quad (\text{iv})$$

specifies the pattern of *Hemato-Finding* objects where the measured parameter is the leukocyte count in \#/mm^3 (i.e., the object pattern (iv) contains two path constraints, one for path *parameter* and one for path *unit*). An example for an object pattern with a constraint on a path with length > 1 is

$$\text{Drug-Administration}[\text{ordered-by.name} = \text{"Taylor, Steve"}]$$

which describes the pattern of *Drug-Administration* objects where the ordering staff member has to have the name "Taylor, Steve".

The *type* of an object pattern is denoted with *Obj-Patt*<*Class*>. For example, the object pattern in (iv) is of type *Obj-Patt*<*Hemato-Finding*>.

Object patterns will play two important roles in this thesis: First, they will be used to specify the requested structure of objects an activity expects as input or which it will provide as output (Chapter 5). Second, a special subtype of object patterns, namely the type *Obj-Patt*<*Activity*> ("activity patterns"), will be used to specify which workflow activities are affected by a control flow failure (Chapter 7).

4.2.1.6 Pattern Subsumption and Pattern Matching

The terms *pattern subsumption* and *pattern matching* deal with the question which patterns are more general than other ones. Both terms play a central role in this thesis, as with them we can define formally which control actions affect which activities of running workflows. We first define pattern subsumption, and second pattern matching.

Pattern subsumption: Let pattern P_1 be of type *Obj-Patt*(*Class*₁) and let P_2 be of type *Obj-Patt*(*Class*₂). We say that P_1 *subsumes* P_2 if any object that fulfills P_2 also fulfills P_1 . Formally, this is given if the following conditions hold:

1. $\text{Class}_2 \text{ IS-A } \text{Class}_1$, and
2. for any object path constraint defined on P_1 , this constraint is fulfilled by any object meeting constraints of P_2 .

For example, if we have

$$P_1 := \text{Drug-Administration}[\text{drug} = \text{"ETOPOSID"}, \text{dosage} > 100, \text{unit} = \text{mg}] \quad (\text{v})$$

$$P_2 := \text{Drug-Administration}[\text{drug} = \text{"ETOPOSID"}, \text{dosage} > 150, \text{unit} = \text{mg}] \quad (\text{vi})$$

then P_1 subsumes P_2 as every ETOPOSID administration with more than 150 mg is also an ETOPOSID

administration of more than 100 mg.

By definition, a pattern P always subsumes itself. Furthermore, we agree on the following terminological convention: If A_1 is of type $Obj-Patt(Class_1)$ and A_2 of type $Obj-Patt(Class_2)$ with $Class_i$ IS-A Activity, and if A_1 subsumes A_2 then we may also say that any A_2 -activity (see 3.3.3) is an A_1 -activity.

Pattern matching: If a pattern P_1 subsumes a pattern P_2 , we may also say that P_2 matches P_1 (as all constraints defined by P_1 are met by P_2). Thus, the *matches* relationship between two patterns can be viewed as the inverse of the *subsumes* relationship.

4.2.1.7 Object-Pattern Matching

The terms *object-pattern matching* deals with the question when an object meets a given pattern. To define this, let P be a pattern of type $Obj-Patt(Class)$, and let o be an object of class $Class$. We say that object o matches P if the following conditions hold:

1. $Class$ IS-A $Class$, and
2. any object path constraint defined by P is fulfilled by o .

For example, if we have

$$P := Drug-Administration[drug = "ETOPOSID", dosage > 100, unit = mg], \quad (vii)$$

$$o:Drug-Administration[drug = "ETOPOSID", dosage = 150, unit = mg], \quad (viii)$$

then o matches P , as a dosage of 150 mg is higher than 100 mg.

4.2.1.8 Object Extensions

For storage purposes, objects can be collected persistently in so-called *object extensions*. For example,

$$extension\ patients(Patient) \quad (ix)$$

defines an extension of *Patient* objects. As the failure handling approach of this thesis abstracts from aspects of data heterogeneity and distribution, extensions such as (ix) do not specify in which data sources the extensions are physically stored. The mapping from F-Logic object extensions to physical data sources such as file systems or relational databases is an implementation matter (see Chapter 11).

In the following, we assume that for a workflow application there exists exactly *one* extension for every F-Logic class. This will facilitate data flow adaptation as described in Chapter 8 (*Structural Adaptation Operators*) significantly. If an object of class $Class$ is inserted into the extension for this class, it immediately is inserted into all extensions for the super-classes of $Class$.

Predicate (for P being a <i>Patient</i> object)	True iff (x = leukocyte count in $\#/mm^3$)
<i>severe-hemato-status</i> (P)	$x < 1000$
<i>critical-hemato-status</i> (P)	$1000 \leq x \leq 2500$
<i>normal-hemato-status</i> (P)	$2500 < x$

Table 4-3: F-Logic predicates for leukocyte ranges under chemotherapy.

4.2.2 F-Logic Predicates

As in first-order predicate logic, *predicates* can be defined over F-Logic objects. Generally, predicates are used to express that some property holds for some objects. For example, Table 4-3 contains three predicates defined on *Patient* objects to describe their hematological status. In AGENTWORK predicates are primarily used to express control actions as introduced in Chapter 3. For example, to realize the control action *drop* used in Chapter 3 we can define the predicate

$$\text{drop}(A, C)$$

with A being of type *Obj-Patt(Activity)* (i.e., an *activity* pattern), and C being an object of class *Case*. This control action states that any activity executed for case C and matching pattern A has to be dropped.

Though predicates syntactically are similar to methods (the “return” type of a predicate is always Bool), predicates differ from methods as they are not assigned to classes to define their behavior, but are defined on objects to express relationships between them.

4.2.2.1 Resource Availability Predicates

A special predicate that will be frequently used describes the availability of resources. Let R be of type *Obj-Patt(Resource)* (i.e., a resource pattern). Then the predicate

$$\text{instances-not-available}(R)$$

returns TRUE when no *Resource* instance exists that matches R .

4.2.3 F-Logic Formulas

F-Logic formulas are statements on objects that are either true or false. As in classical logic [SCHÖNING 1989], we can define F-Logic formulas inductively as follows:

- If o_1, o_2, \dots, o_n are F-Logic objects and P is a predicate defined on them, then $P(o_1, o_2, \dots, o_n)$ is a formula.

- If F is a formula, then $NOT\ F$ is a formula.
- If F, G are formulas, then $F\ AND\ G, F\ OR\ G$ are formulas.
- If o is an object variable, and G a formula, then
 - $EXISTS\ o\ G$ (“it exists an object o so that G is true”) and
 - $FOR-ALL\ o\ G$ (“for all objects o , G is true”) are formulas.

4.2.4 F-Logic Queries

F-Logic queries are expressed with the “?-” operator, e.g.,

?- $P\ IN\ patients\ AND\ P.diagnosis = "Hodgkin-Lymphoma"$
 $AND\ P.doctor.name = "Taylor, Steve"$.

This query selects all objects from the extension *patients* with diagnosis “Hodgkin-Lymphoma” who are treated by a doctor with name “Taylor, Steve”.

Note that F-Logic queries always are *retrieval* queries. Object insertion, removal and updating has to be done by extension operators which we do not discuss in detail here.

4.2.5 F-Logic Rules

Rules in F-Logic are used to express which formulas imply other formulas. For example, if A denotes the administration of ETOPOSID, then the rule

$WHEN\ severe-hemato-status(P)$ (x)
 $THEN\ drop(A, P).$

states that whenever a patient P has a severe hematological status, that then ETOPOSID has to be dropped for P . Note that a rule such as (x) is not an ECA rule as it has no notion of “events”. It only states that the formula $drop(A, P)$ becomes true when another formula $severe-hemato-status(P)$ becomes true. In particular, it does *not* state how a formula such as $severe-hemato-status(P)$ relates to events such as inserting a new object containing hematological data into some object extension. This will be described in Section 4.4, where the ACTIVETFL notion of events and *active* rules on events is explained.

4.3 Temporal Frame Logic

We now describe our temporal extension of F-Logic. For this, we introduce important prerequisites, namely *temporal frames*, *temporal durations* and *distances*, and *temporal functions*. Then, the central concept of *temporal formulas* is introduced which allows to assign fixed or conditional valid times to F-Logic formulas.

4.3.1 Temporal Frames

A temporal frame $TF = (T, <)$ is a tuple consisting of a non-empty discrete set T of “points in time” (i.e., the “time axis”), ordered by a non-reflexive binary relation $<$ of precedence (“earlier than”) [BENTHEM 1995]. As for the domains addressed by AGENTWORK the time does not have to go back to the past unlimitedly, we can assume a minimum of T , i.e., that a $t_{min} \in T$ exists with $t_{min} < t$ for all $t \in T$ with $t \neq t_{min}$.

A typical set of points in time is the set of calendar points in time of the *gregorian* calendar starting from 1 Jan 0000 or 1 Jan 1900. For example, for HEMATOWORK we can use

$$T = \{1 \text{ Jan } 1900: 0.0 \text{ am}, \dots, 1 \text{ Jan } 2001: 0.0 \text{ am}, 1 \text{ Jan } 2001: 0.1 \text{ am}, \dots, 1 \text{ Jan } 2001: 0.59 \text{ am}\}^5 (xi)$$

if we assume that no patient has been born before 1900 and that it is sufficient to take the hour as the smallest unit of time⁶. In the following we restrict ourselves to points in time such as (xi) , i.e., points in time based on the gregorian calendar with the smallest granularity *hour* as this is sufficient for the application domains addressed by AGENTWORK.

4.3.2 Temporal Durations and Distances

Let $TF = (T, <)$ be a temporal frame based on the gregorian calendar. A temporal *duration* is a tuple (v, u) with $v \geq 0$ (duration value) and $u \in \{hour, day, week, month, year\}$ (duration unit).

Such a tuple (v, u) may alternatively be called temporal *distance* (instead of duration), if it describes the amount of time that passes when time shifts from a point in time t_1 to a point in time t_2 with $t_1 < t_2$. For example, the temporal distance between the points in time $1 \text{ Jan } 1900: 0 \text{ am}$ and $1 \text{ Jan } 1900: 7 \text{ am}$ is $(7, hour)$. In the following, we will consider only sets of gregorian points in time where the distance between two subsequent points in time always is the same. For example, for T as defined in (xi) the distance between two subsequent points in time always is $(1, hour)$.

If T is a set of points in time with such uniform distances between two subsequent points in time, a duration is said to be *T-conform* if it always is a whole-numbered multiple of the distance between two subsequent points in time of T . For example, the durations $(2, hour)$, $(0.5, day)$, and $(2, day)$ are conform to T as defined in (xi) , as $(0.5, day) = (12, hour)$ and $(2, day) = (48, hour)$. In contrast to this, the durations $(0.5, hour)$ and $(0.7, day) = (16.8, hour)$ are not *T-conform*.

For temporal frames consisting of more general sets of points in time and more general distance metrics we refer to [BENTHEM 1995, MANNA & PNUELI 1992].

4.3.3 Temporal Functions

Let again $TF = (T, <)$ denote a gregorian temporal frame with uniform distances and with the small-

5. $x.y \text{ am}$ denotes the x -th hour and y -th minute am.

6. In disciplines such as intensive care one may have to model time on the level of seconds.

Function	Arguments	Return value	Remarks
$succ(t)$	$t \in T$	Direct successor point in time of t	Alternative notations: $t + 1$ for $succ(t)$ $t - 1$ for $pred(t)$
$pred(t)$	$t \in T, t \neq t_{min}$	Direct predecessor point in time of t	
$t + n$	$t \in T, n \in \mathbb{N}_0$	Defined inductively: $t + 1 := succ(t)$ $t + n := succ(t + n - 1)$	
$t - n$	$t \in T, n \in \mathbb{N}_0$ with $t \geq t_{min} + n$	Defined inductively: $t - 1 := pred(t)$ $t - n := pred(t - n + 1)$	
$ t_1 - t_2 $	$t_1, t_2 \in T$	Distance between t_1 and t_2	$ t_1 - t_2 = t_2 - t_1 $
$t + (v, u)$	$t \in T, (v, u) = T\text{-conform duration}$	Point in time $t' \geq t$ with $ t' - t = (v, u)$	
$t - (v, u)$	$t \in T, (v, u) = T\text{-conform duration with}$ $t \geq t_{min} + (v, u)$	Point in time $t' \leq t$ with $ t' - t = (v, u)$	
$(v_1, u) + (v_2, u)$	$(v_1, u), (v_2, u)$	$(v_1 + v_2, u)$	
$(v_1, u) - (v_2, u)$	$(v_1, u), (v_2, u)$ $= T\text{-conform durations}$	$(v_1 - v_2, u)$	

Table 4-4: Temporal functions.

est granularity *hour*. Table 4-4 lists several temporal functions that will be frequently used in this thesis. Other functions not listed in the table may deal with the transformation of temporal statements from one granularity level to another. For example, a function $days2hours(v, day)$ may transform a temporal duration or distance in days to a notation in hours, e.g., $days2hours(4, day) = (96, hour)$.

4.3.4 Temporal Formulas

So far, an F-Logic formula such as

$$drop(A, bob)$$

does not state *when* it is valid, i.e., for how long the activity specified by A shall be dropped for bob (e.g., whether it shall be dropped only for “the moment” or “for a while” or “for ever”). To express this ACTIVETFL allows to assign either a fixed or conditional valid time to a formula. In the following, let again $TF = (T, <)$ denote a gregorian temporal frame with uniform distances between two subsequent points in time.

4.3.4.1 Fixed Valid Time in ACTIVETFL

A fixed valid time is any set $S \subset T$ which is described either by an explicit listing of points in time

or by temporal functions. Examples for fixed valid time sets include

- $[2 \text{ Dec } 2002: 8 \text{ pm}, 4 \text{ Dec } 2002: 6 \text{ pm}]$ = The set of points in time starting at $2 \text{ Dec } 2002: 8 \text{ pm}$ and ending at $4 \text{ Dec } 2002: 6 \text{ pm}$ (by using the interval notation “ $[\]$ ” as a shortcut for the listing of all points in time between these two points in time).
- $[2 \text{ Dec } 2002: 8 \text{ pm}, 2 \text{ Dec } 2002: 8 \text{ pm} + (72, \text{ hour})]$ = The set of points in time starting at $2 \text{ Dec } 2002: 8 \text{ pm}$ and ending after 72 hours (i.e., at $5 \text{ Dec } 2002: 8 \text{ pm}$).
- $[\text{now}, \text{now} + (72, \text{ hour})]$ = The set of points in time starting at the current system time now (rounded to the closest point in time of T) and ending after 72 hours.
- $[t, \infty)$ = The set of all points $t' \geq t$ (∞ being the symbol for “ad infinitum”).
- Any combination of fixed valid time sets constructed via the set operators \cup , \cap or \setminus (the latter symbol denoting the complement set).

Such a fixed valid time S then can be assigned to any formula F via the *VALID-TIME* statement, i.e.,

$F \text{ VALID-TIME } S$

states that F holds at every $t \in S$. For example,

$\text{drop}(A, \text{bob}) \text{ VALID-TIME } [2 \text{ Dec } 2002: 8 \text{ pm}, 4 \text{ Dec } 2002: 6 \text{ pm}]$ (xii)

states that the activity specified by A shall be dropped for bob from $2 \text{ Dec } 2002: 8 \text{ pm}$ until $4 \text{ Dec } 2002: 6 \text{ pm}$. An example for a rule with such a *VALID-TIME* statement is

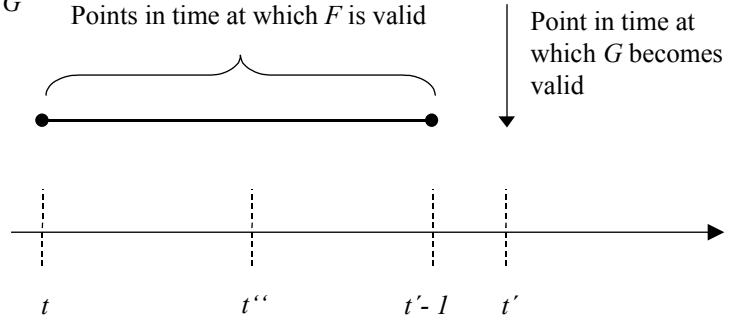
$\text{WHEN } \text{severe-hemato-status}(P) \text{ VALID-TIME } [\text{now} - (7, \text{ day}), \text{now}]$ (xiii)
 $\text{THEN } \text{drop}(A, P) \text{ VALID-TIME } [\text{now}, \text{now} + (4, \text{ day})].$

This rule states that whenever the predicate $\text{severe-hemato-status}(P)$ is valid for the last 7 days, that then $\text{drop}(A, P)$ is valid for the next four days (the way how $\text{WHEN } \text{severe-hemato-status}(P) \text{ VALID-TIME } [\text{now} - (7, \text{ day}), \text{now}]$ can be related to events is described in 4.4). If no fixed valid time has been assigned to a formula in the *WHEN* or *THEN* part the valid time is assumed to be now . For example, if rule (xiii) would not consist of any *VALID-TIME* statement this would mean that whenever the predicate $\text{severe-hemato-status}(P)$ holds, that then $\text{drop}(A, P)$ holds exactly for the point in time at which the rule has been triggered (e.g., for one hour when taking (xi) as temporal frame).

Note that (xii) and (xiii) are reified logical expressions and are equivalent with the *Holds* notation in 2.5.3 (*Temporal Reasoning*), e.g., (xii) could also be expressed as

$\text{Holds}(\text{drop}(A, \text{bob}), [2 \text{ Dec } 2002: 8 \text{ pm}, 4 \text{ Dec } 2002: 6 \text{ pm}])$

Due to its better readability, the *VALID-TIME* notation is preferred in this thesis.

Figure 4-3: Illustration of $F \text{ Until } G$ formula.


4.3.5 Conditional Valid Time in ACTIVETFL

To describe a valid time conditionally by a termination condition, ACTIVETFL provides the two temporal operators *Until* and *Unless* which have been introduced by several temporal logics [CHOMICKI & TOMAN 1998, MANNA & PNUELI 1992]. With these two operators it can be stated in what way the valid time of an F-Logic formula is related to the valid time of another formula. In particular, we will show that temporal dependencies such as the one introduced in Section 4.1.3.2 can be represented better by these temporal operators than by SQL triggers. Other temporal operators known from temporal logics – such as *Eventually*, *Has-Always-Been*, *Once* and *Back-To* – are not required for handling control flow failures so that they are not described in this thesis. In the following, F and G are F-Logic formulas while t, t', t'' denote points in time.

Until operator: This operator is used to express that a formula G eventually will be valid in the future and that a formula F is valid at least until G (first) becomes valid (Figure 4-3), i.e.,

It holds:		it holds:
$(F \text{ Until } G) \text{ VALID-TIME } t$	iff	It exists $t' > t$: $G \text{ VALID-TIME } t'$ and for all t'' with $t \leq t'' < t'$: $F \text{ VALID-TIME } t''$ AND $\text{NOT } (G \text{ VALID-TIME } t'')$

Note that the definition of $F \text{ Until } G$ does not exclude that F also holds from t' on. Furthermore, if it holds $(F \text{ Until } G) \text{ VALID-TIME } t$ and if $t' > t$ is the first point in time for which G becomes valid, then it also holds:

$$(F \text{ Until } G) \text{ VALID-TIME } t'' \text{ for } t \leq t'' < t'. \quad (\text{xiv})$$

A typical medical example for the *Until* operator is the rule

$$\begin{aligned} \text{WHEN } & \text{critical-hemato-status}(P) \text{ VALID-TIME } [\text{now} - (7, \text{day}), \text{now}] & (\text{xv}) \\ \text{THEN } & (\text{add-repetitively}(\text{DOXYCYCLIN}, (1, \text{day}), P) \text{ Until drop}(\text{ETOPOSID}, P)) \\ & \text{VALID-TIME now} \end{aligned}$$

This rule states that whenever a patient P has a critical hematological status during the last 7 days, P must get the drug DOXYCYCLIN every day until the drug ETOPOSID is dropped. The medical background for this example is that a cytostatic drug such as ETOPOSID significantly increases the probability of serious bacterial infections because of its immune-suppressive side-effects. Therefore, antibiotic drugs such as DOXYCYCLIN are given prophylactically during chemotherapy when the hematological situation becomes critical.

If the valid time of an *Until* formula in the *THEN* part of a rule is *now* – such as in (xv) – it can also be omitted, i.e., the *THEN* part of (xv) may be also written as

$$\text{THEN} \quad \text{add-repetitively}(\text{DOXYCYCLIN}, (1, \text{day}), P) \text{ Until drop}(\text{ETOPOSID}, P) \quad (\text{xvi})$$

with the meaning that $\text{add-repetitively}(\text{DOXYCYCLIN}, (1, \text{day}), P) \text{ Until drop}(\text{ETOPOSID}, P)$ holds at the point in time when the rule has been triggered (i.e., at *now*).

Unless (Waiting-for) operator: A limitation of $F \text{ Until } G$ is that it requires that G will eventually occur. In the example of rule (xv) this is no problem as a drug such as ETOPOSID cannot be given arbitrarily long due to its strong toxicity (i.e., the point in time where $\text{drop}(\text{ETOPOSID}, P)$ becomes valid will definitely occur). However, often weaker statements are required stating that, at a point in time t , F is valid either until G becomes valid, or is valid forever in case that G will never become valid in the future. This is done by the *Unless* operator. Formally, *Unless* is defined as

It holds:		it holds:
$(F \text{ Unless } G) \text{ VALID-TIME } t$	iff	$(F \text{ Until } G) \text{ VALID-TIME } t \text{ OR }$
(at point in time t , F is valid		$F \text{ VALID-TIME } [t, \infty)$
unless G is valid)		

With the *Unless* operator, we can now express dependencies such as the one described in Section 4.1.3.2, i.e., that the drug ETOPOSID has to be dropped when a patient has a severe hematological status for the last two days (leukocyte count < 1000) and that ETOPOSID can only be given again when the hematological status becomes normal again (leukocyte count > 2500):

$$\begin{aligned} &\text{WHEN severe-hemato-status}(P) \text{ VALID-TIME } [\text{now} - (2, \text{day}), \text{now}] \\ &\text{THEN drop}(\text{ETOPOSID}, P) \text{ Unless normal-hemato-status}(P) \text{ VALID-TIME now} \end{aligned} \quad (\text{xvii})$$

This notation first is more compact than the trigger notation in Table 4-1. Second, negative side-effects between different rules can be avoided. Concerning the latter point, let us assume that there is another rule stating that ETOPOSID should also be dropped when a patient has a severe *renal* status since two days and that ETOPOSID can only be given again when the renal status becomes normal again (e.g., when the creatinin clearance is higher than 80 again):

$$\begin{aligned} &\text{WHEN severe-renal-status}(P) \text{ VALID-TIME } [\text{now} - (2, \text{day}), \text{now}] \\ &\text{THEN drop}(\text{ETOPOSID}, P) \text{ Unless normal-renal-status}(P) \text{ VALID-TIME now} \end{aligned} \quad (\text{xviii})$$

Furthermore, let us assume for a sequence t_0, t_1, \dots, t_7 of points in time that rule (xvii) is triggered at t_1 and that rule (xviii) is triggered at t_3 (Table 4-5). Then, when the hematological status becomes normal again for t_5, t_6 and t_7 , then $drop(ETOPOSID, P)$ is still valid at t_5 and t_6 as $drop(ETOPOSID, P)$ Unless normal-renal-status(P) is valid at t_5 and t_6 (as t_7 is the first point in time where the renal status becomes normal again). Therefore, side effects such as that the termination condition for dropping ETOPOSID in case of critical leukocyte counts also may terminate the dropping of ETOPOSID in case of a renal toxicity cannot occur (in contrast to the SQL trigger solution in Table 4-1).

4.4 Active Rules

We now describe the AGENTWORK notion of *active* rules. Principally, active rules are rules such as those described in 4.2 and 4.3, but with the following additional characteristics.

- First, an explicit notion of primitive and composite “events“ is added.
- Second, a *WITH* part is added by which conditions can be specified that filter *relevant* events.
- Third, the *THEN* part is restricted to *control actions*, i.e., F-Logic formulas stating what has to be done concerning workflows or activities.

This section describes only the principal structure of active rules in AGENTWORK. Advanced topics such as rule integrity are described in Chapter 7 after the full list of control actions has been introduced. In the following, $TF = (T, <)$ again denotes a gregorian temporal frame.

4.4.1 Events

In ACTIVETFL, an *event* is something that happens at a point in time. We first describe so-called primitive events and conditions on primitive events. Second, we describe the construction of composite events from other events. Third, we introduce conditions for composite events. Fourth, we discuss several event consumption policies stating to how many occurrences of a composite event an event may contribute. Fifth, the relationship between predicate formulas and composite events is described.

4.4.1.1 Primitive Events and Conditions

In ACTIVETFL, a *primitive* event is the occurrence of a basic operation on an extension (see 4.2.1.8) storing objects of class *Event*. We recall from Chapter 3 that objects of class *Event* describe events that occur to a case such as a patient (such as a hematological event described by the *Event* subclass *Hemato-Finding*; see Figure 4-2) or to a resource, and that such events have the competence to trigger control flow failures. Thus, any insertion, removing or updating of such an *Event* object in the respective object extension is a relevant event that may have to be monitored by active rules. For this, ACTIVETFL supports the primitive event types INSERT, REMOVE and UPDATE corresponding to the respective operations on extensions of *Event* objects. For example, for the extension

hemato-findings(*Hemato-Finding*)

of *Hemato-Finding* objects the line

WHEN INSERT ON hemato-findings

specifies the primitive event of inserting a *Hemato-Finding* object into the extension *hemato-findings*. Please recall from 4.2.1.8 that object extensions such as *hemato-findings* may cover data from any data source such as databases, file systems, or user interfaces. In particular, AGENTWORK assumes that for any new or updated data item describing a case or resource event there is at least one INSERT or UPDATE event bringing this data item into at least one extension of *Event* objects. Other event types as described for instance by [WIDOM & CERI 1996, PATON 1999] – such as transaction events (e.g., transaction abort) – are not considered as they typically do not cause *control flow* failures. However, if necessary ACTIVETFL can easily be extended with such other event types as they do not imply any additional complexity for active rules.

Furthermore, as ACTIVETFL addresses the control flow adequacy of workflows and *not* data integrity aspects, ACTIVETFL rules do not distinguish between the state *before* and *after* an insert, update or remove event (in contrast to SQL triggers). Rather, ACTIVETFL implicitly considers only rules triggered *after* an insert, update or deletion event. Rules triggered immediately *before* such an event would be needed if for example the insertion of data violating any integrity constraints shall

be avoided. As this is out of scope of this thesis, such “before“-triggers are not considered.

To filter relevant events, a *condition* can be assigned to a primitive event in the *WITH* part of a rule. This condition may consist of any formula f on the data object referenced by the event in the *WHEN* part. For referencing purposes, two symbols *new* and *old* are provided. For an INSERT event, *new* denotes the new object inserted. For an UPDATE event such as the change of an attribute value, *old* denotes the old version of the object before the update while *new* denotes the new version after the update. For a REMOVE event, *old* denotes the object removed from the extension⁷. An ACTIVETFL then rule is said to be triggered at the point in time t if the event described in the *WHEN* part occurs at t and if the formula f in the *WITH* part is valid at t .

An example for a primitive event with a condition is

(xix)

WHEN INSERT ON hemato-findings
WITH new.parameter = Leukocyte-Count AND new.value < 1000.

In the following, the term *primitive event* denotes any INSERT, UPDATE or REMOVE event with or without a condition.

4.4.1.2 Composite Events

Furthermore, so-called *composite* events can be defined. ACTIVETFL supports the composite event types *conjunction*, *disjunction*, *negation*, *sequence*, and *time series*, as introduced by [GEHANI ET AL. 1992, CHAKRAVARTHY ET AL. 1994, MOTAKIS & ZANIOLO 1997 A].

In the following, E, E_1, E_2, \dots, E_n first will denote primitive events while $I \subset T$ denotes a non-empty fixed set of points. Furthermore, the occurrence of an event type will also be termed as an *instance* of an event type.

Conjunction: The conjunction of E_1, E_2, \dots, E_n – written as $CONJ(E_1, E_2, \dots, E_n, I)$ – occurs during I if *all* E_i occur during I . Formally,

$CONJ(E_1, E_2, \dots, E_n, I)$ occurs	iff	for all $i \in \{1, \dots, n\}$ it exists a $t_i \in I$ at which E_i occurs	(xx)
--	-----	--	------

The point in time at which an instance of $CONJ(E_1, E_2, \dots, E_n, I)$ occurs is the point in time at which the last E_i occurs that is needed to establish this instance. For example, if $I = \{t_1, t_2, t_3, t_4\}$ and E_2 occurs at t_1 and E_1 at t_3 , then the instance of $CONJ(E_1, E_2, I)$ established by these occurrences of E_1, E_2 is said to occur at t_3 . By restricting I to a single point in time it can be expressed that the E_1, E_2, \dots, E_n have to occur simultaneously. Note that the question how often $CONJ(E_1, E_2, \dots, E_n, I)$ may occur during I depends on the event consumption policy selected (see 4.4.1.4 below).

7. The question whether an object removed from an extension is also physically deleted is viewed as an implementation issue.

Disjunction: The disjunction of E_1, E_2, \dots, E_n – written as $DISJ(E_1, E_2, \dots, E_n, I)$ – occurs during I if at least one E_i occurs during I . Formally,

$$DISJ(E_1, E_2, \dots, E_n, I) \text{ occurs} \quad \text{iff} \quad \text{it exists an } i \in \{1, \dots, n\} \text{ for which an } t_i \in I \text{ exists at which } E_i \text{ occurs} \quad (xxi)$$

The point in time at which an instance of $DISJ(E_1, E_2, \dots, E_n, I)$ occurs is the point in time at which the E_i occurred that established this instance. For example, if $I = \{t_1, t_2, t_3, t_4\}$ and E_1 occurs at t_2 and E_2 at t_3 , then the instance of $DISJ(E_1, E_2, I)$ established by E_1 is said to occur at t_2 while the instance of $DISJ(E_1, E_2, I)$ established by E_2 is said to occur at t_3 .

Negation: The negation of an event E – written as $NEG(E, I)$ – occurs during I if E does *not* occur during I . Formally,

$$NEG(E, I) \text{ occurs} \quad \text{iff} \quad \text{there is no } t \in I \text{ at which } E \text{ occurs} \quad (xxii)$$

The point in time at which $NEG(E, I)$ occurs is the last $t \in I$. For example, if $I = \{t_1, t_2, t_3, t_4\}$ and E occurs at none of these four points in time, then $NEG(E, I)$ occurs at t_4 .

Sequence: A sequence of events E_1, E_2, \dots, E_n – written as $SEQ(E_1, E_2, \dots, E_n, I)$ – occurs during I if the E_i occur sequentially during I . Formally,

$$SEQ(E_1, E_2, \dots, E_n, I) \text{ occurs} \quad \text{iff} \quad \text{it exist } t_1 < t_2 < \dots < t_n, t_i \in I \text{ with } E_i \text{ occurring at } t_i \quad (xxiii)$$

The point in time at which an instance of $SEQ(E_1, E_2, \dots, E_n, I)$ occurs is the point in time at which E_n occurs. For example, if $I = \{t_1, t_2, t_3, t_4\}$ and E_1 occurs at t_1 and E_2 at t_3 , then the instance of $SEQ(E_1, E_2, I)$ established by these occurrences of E_1, E_2 is said to occur at t_3 .

Time Series: Time series are a special type of sequence events that are of particular importance for medical domains. For example, a single critical finding such as a low leukocyte value does not necessarily induce some control flow failure. Rather, often only the *repetitive* occurrence of critical findings may induce a control flow failure. For this, ACTIVETFL provides *time series events*: Given some event E , a time series event over E with length n , minimal and maximal distances d_{min} and d_{max} occurs during I , if E occurs repetitively during I at a sequence of n points in time with a

minimal distance of d_{min} and a maximal distance of d_{max} between two successive points of the sequence. Formally,

$$TIME-SERIES(E, n, d_{min}, d_{max}, I) \text{ occurs} \quad \text{iff} \quad \begin{array}{l} \text{it exist } t_1 < t_2 < \dots < t_n, t_i \in I \text{ with:} \\ \bullet d_{min} \leq |t_i - t_{i-1}| \leq d_{max}, i = 2, \dots, n \\ \bullet E \text{ occurs at every } t_i \end{array} \quad (xxiv)$$

The point in time at which an instance of $TIME-SERIES(E, n, d_{min}, d_{max}, I)$ occurs is t_n (as then the last instance of E establishing the time series occurred).

A typical medical example for a time series event is the following: Let E be the event that the leukocyte count of a patient is less than 1000 as defined in (xix). Then, the composite event

$$WHEN TIME-SERIES(E, 3, (2, day), (4, day), [now, now + (2, week)]) \quad (xxv)$$

occurs if during the next two weeks the leukocyte count of a patient is less than 1000 at 3 points in time with a minimal distance of 2 days and a maximal distance of 4 days between two leukocyte measurements.

The specification of a minimal or maximal distance d_{min} respective d_{max} between the sequence points makes sense as often occurrences of E being too close together or too far away from each other have a limited significance. For example, on one side two leukocyte count measurements at two subsequent days do not mean more information than one measurement, as the leukocyte value usually does not change significantly during two days. On the other side, two leukocyte counts l_1 and l_2 being too far away from each other bear the risk that between them the leukocyte count has been totally different than l_1 and l_2 . Thus, a physician often will be only interested in a time series event with a minimal and maximal distance between two measurements, such as specified in (xxv). If only d_{max} or only d_{min} has been specified, the sequence only has to meet $|t_i - t_{i+l}| \leq d_{max}$ respective $d_{min} \leq |t_i - t_{i+l}|$.

Recently, time series have gained additional attention in the fields of medical informatics and also in data mining. In these fields, several other types of time series have been defined which are not introduced in this thesis, as we want to focus not on composite events such as time series but on the handling of control flow failures. Therefore, for more extended approaches concerning time series events, we refer to [KLOUDAS ET AL. 2000, BELAZZI ET AL. 1999, BERNDT & CLIFFORD 1996].

As for any composite event the point in time of its occurrence is precisely defined, the definitions (xx) – (xxiv) also hold if the events establishing the composite event themselves are composite events.

4.4.1.3 Composite Events with Conditions

Often it is necessary for a composite event established by primitive events to define a condition which cannot be expressed as a number of conditions w.r.t. the primitive events. For example, in

case of the leukocyte time series event in (xxv) it has to be specified that all occurrences of E establishing the time series have to be related to the *same* patient. Furthermore, the physician additionally may want to specify the condition that the measurements also show a decreasing tendency which is an even worse clinical situation than leukocyte counts being only smaller than 1000. Such a composite condition may consist of any formula f on the data objects referenced by the primitive events establishing the composite event. For referencing purposes, the symbols new_i and old_i are used. The index i refers to the i -th event in the composite event expression $CONJ(E_1, E_2, \dots, E_n, I)$, $DISJ(E_1, E_2, \dots, E_n, I)$ or $SEQ(E_1, E_2, \dots, E_n, I)$, or to the i -th occurrence of E in case of $TIME-SERIES(E, n, d_{min}, d_{max}, I)$. In case of $NEG(E, I)$, no symbol new is needed as E does not occur during I . The symbol new_i is provided in case the referenced primitive event is an INSERT or UPDATE. The symbol old_i is provided in case the referenced primitive event is an UPDATE or REMOVE. With these symbols, our decreasing leukocyte trend could be defined as

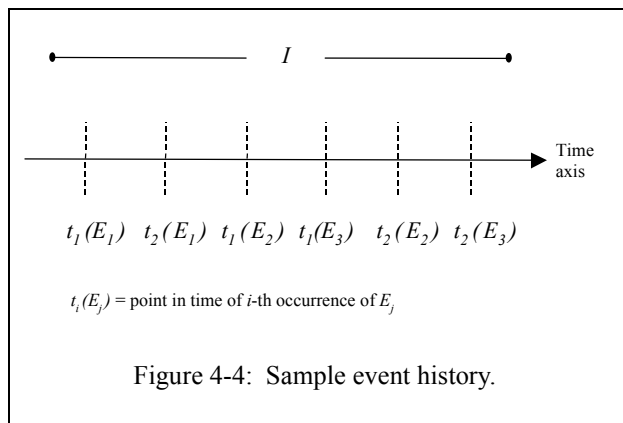
$$\begin{aligned} & \text{WHEN TIME-SERIES}(E, 3, (2, \text{day}), (4, \text{day}), [\text{now}, \text{now} + (2, \text{week})]) \quad (\text{xxvi}) \\ & \text{WITH } new_1.of = new_2.of \text{ AND } new_2.of = new_3.of \text{ AND} \\ & \quad new_1.value > new_2.value \text{ AND } new_2.value > new_3.value \end{aligned}$$

Please recall that in medical applications *of* refers to the *Patient* object to which the event occurs (see 3.2.1 and 4.2.1.2). As it is self-evident that for medical composite events all establishing events refer to the same patient, we omit conditions such as $new_1.of = new_2.of$ in the following examples. In particular, as a shortcut we use P_E or P_{E_i} to denote the patient to which an event E or E_i occurred.

If necessary, this referencing mechanism can be extended also for larger composition hierarchies by a multi-dimensional index. For example, if we have a composite event constructed from composite events constructed from primitive events, then $new_{i,j}$ refers to the i -th primitive event of the j -th composite event establishing the top level composite event.

4.4.1.4 Event Consumption Policies

Consumption policies state the way in which events may contribute to a composite event [PATON 1999, MOTAKIS & ZANIOLO 1997 B, CHAKRAVARTHY ET AL. 1994]. The appropriate policy highly depends on the application domain. As this thesis focuses on medical workflow applications, we restrict ourselves to the policies relevant for this application area, namely the policies *unrestricted*, *chronicle*, and *continuous*. For other policies, such as *recent* and *cumulative*, we refer to [PATON 1999, MOTAKIS & ZANIOLO 1997 B]. To illustrate the following poli-



cies, we use an example given in [CHAKRAVARTHY ET AL. 1994]. This example assumes a sequence composite event $SEQ(E_1, E_2, E_3, I)$ of some primitive events E_1, E_2, E_3 which occur in the relative order shown in Figure 4-4 ($t_i(E_j)$ denotes the point in time of the i -th occurrence of E_j).

Unrestricted Policy: With this policy, *any* event occurrence combination that matches the definition of the composite event establishes the occurrence of this composite event. In Figure 4-4, occurrences of $SEQ(E_1, E_2, E_3, I)$ will be established at

$$\{t_1(E_1), t_1(E_2), t_1(E_3)\}, \{t_1(E_1), t_1(E_2), t_2(E_3)\}, \{t_1(E_1), t_2(E_2), t_2(E_3)\}, \\ \{t_2(E_1), t_1(E_2), t_1(E_3)\}, \{t_2(E_1), t_1(E_2), t_2(E_3)\} \text{ and } \{t_2(E_1), t_2(E_2), t_2(E_3)\}.$$

Chronicle Policy: With this policy, events contribute to a composite event in their *chronological* order of occurrence. Only the oldest occurrence, respectively, of an event is used. When a composite event is established, the used occurrences of the establishing events cannot participate in any other occurrences of this composite event (of course, they can still be used for other composite event types). In Figure 4-4, with the chronicle policy occurrences of $SEQ(E_1, E_2, E_3, I)$ will be established at

$$\{t_1(E_1), t_1(E_2), t_1(E_3)\} \text{ (these occurrences are then removed from the event history) and } \\ \{t_2(E_1), t_2(E_2), t_2(E_3)\}.$$

The chronicle policy is suitable for applications where some causal dependency exists between the events establishing the composite event. For example, let us assume that in a doctor's ambulance E_1 is the problem report of a patient, E_2 the diagnosis event and E_3 the prescription of some drug regarding the same patient. Assuming furthermore that the patients are handled by a FIFO⁸ strategy in the ambulance, then only the chronicle sequences $\{t_1(E_1), t_1(E_2), t_1(E_3)\}$ and $\{t_2(E_1), t_2(E_2), t_2(E_3)\}$ are of interest as each describes the handling of *one* patient. The composite event then may be raised to generate the bill for the patient.

Continuous Policy: This policy checks at every point in time which composite events have occurred w.r.t. a given composite event type. Then, all establishing events of these composite events are removed from the event history for this composite event type, so that they cannot contribute to other occurrences of this composite event type anymore. By this policy some sort of a sliding point is defined which moves from the "left to the right" of the time axis, and for which all events at the left side that already contributed to a given composite event type are not considered anymore for this type. In Figure 4-4, the first point in time where occurrences of $SEQ(E_1, E_2, E_3, I)$ will be established is $t_1(E_3)$, as $SEQ(E_1, E_2, E_3, I)$ is raised by

$$\{t_1(E_1), t_1(E_2), t_1(E_3)\} \text{ and } \{t_2(E_1), t_1(E_2), t_1(E_3)\}.$$

8. First In First Out

Then, with the continuous policy, the occurrences of E_1 , E_2 , and E_3 at $t_1(E_1)$, $t_2(E_1)$, $t_1(E_2)$ and $t_1(E_3)$ are removed from the event history for $SEQ(E_1, E_2, E_3, I)$ with the consequence that there is no more occurrence of $SEQ(E_1, E_2, E_3, I)$.

The continuous policy is suitable for trend detection problems. For example, when having detected decreasing leukocyte counts by (xxvi), the physician usually wants to see whether his procedures undertaken to reverse this trend have been successful. This can best be done by inspecting *new* leukocyte counts so that the old leukocyte counts raising (xxvi) beforehand are of no further interest.

The particular policy that holds for a composite event definition is expressed by the keywords POLICY UNRESTRICTED, POLICY CHRONICLE, and POLICY CONTINUOUS, e.g.,

WHEN TIME-SERIES(E, 3, (2, day), (4, day), [now, now + (2, week)])
POLICY CONTINUOUS
WITH new1.value > new2.value AND new2.value > new3.value

4.4.1.5 Events and Predicates

As already sketched in former sections of this chapter, a predicate formula such as

severe-hemato-status(P) VALID-TIME [t, t + d], t ∈ T, d some duration (xxvii)

does not make any statement how it is related to data-oriented events such as the insertion of *Hemato-Finding* objects into some data source. To close this gap, we can use composite events to define such predicates in a more data-oriented manner than it has been done in Table 4-3. For example, for E being the event defined in (xix) we could define that

severe-hemato-status(P) VALID-TIME [t, t + (1, week)]

holds iff

TIME-SERIES(E, 3, $d_{min} = (48, \text{hour})$, [t, t + (1, week)]), P_E = P

occurs, i.e., iff there are three leukocyte count measurements during this week which are less than 1000 and which have a temporal distance of at least 48 hours between two subsequent measurements.

4.4.2 Actions

The *THEN* part of an active rule consists of exactly one control action, e.g.,

WHEN TIME-SERIES(E, 3, (2, day), (4, day) [now - (2, week), now] (xxviii)
POLICY CONTINUOUS
WITH new1.value > new2.value AND new2.value > new3.value
THEN drop(ETOPOSID, P_E) VALID-TIME [now, now + (1, week)]

with E being the event defined in (xix) (and P_E being the patient w.r.t. whom the three instances of E occurred). In particular, the *THEN* part of an ACTIVETFL rule must not contain any data manipulation statements or procedure calls. This restrictive structure of *THEN* parts is because active rules in AGENTWORK shall specify what shall be done with a failed workflow on a high level of abstraction. Any details such as the order of control actions (in case that multiple rules such as (xxviii) are triggered) and the nodes that may have to be dropped or added, can only be determined at execution time when the particular structure of the failed workflow is known. Furthermore, the restrictive structure of the *THEN* part supports rule integrity as we will show in Chapter 7 (*Control Actions*). Note that the restriction to one control action is only syntactically and does not reduce expressiveness w.r.t. control actions. For example, if for a single event E two control actions have to be triggered, this can be expressed by *two* active rules which are both triggered by E and where each rule triggers one of the two control actions.

As our active rules are clearly related to the handling of control flow failures, we call them also rules for control flow failures in the following (as already done in Chapter 3). The complete listing of control actions supported by ACTIVETFL will be given in Chapter 7.

4.5 Summary

In this chapter, we motivated and introduced ACTIVETFL as the data and rule definition language for AGENTWORK. This language has been based on F-Logic, to which elements of a temporal logic have been added. Furthermore, a model of primitive and composite events has been added to cover a broad range of events that may occur in domains such as hemato-oncology and that may cause control flow failures. The main reason for selecting a *temporal logic* as the core of ACTIVETFL has been that such a temporal logic provides a better support of temporal requirements than for example SQL:99 or ODL/OQL. In particular, we have shown that temporal logic operators such as *Unless* allow to express failure rules where the valid time of a control action depends on the occurrence of other events in a compact and clear manner.

We conclude the chapter by emphasizing two important points: First, ACTIVETFL is *not* a general purpose language. In particular, the *THEN* part of active rules has been restricted significantly (if compared with general active rules) as it contains exactly one control action. This has been done to reduce the language complexity⁹, to support rule integrity (see Chapter 7) and to facilitate the handling of control flow failures. Note that such a restriction is not possible for the *WHEN* part as composite events are needed for many application classes and as composition types such as event conjunction cannot be expressed by “decomposing” and “distributing” the event among different rules. Second, ACTIVETFL allows for the definition of data and rules on a high level of abstraction. It is the matter of an implementation such as the one described in Chapter 11 to decide which languages (e.g., one of the prototypical F-Logic implementations, SQL:99, or PROLOG) should be used to implement the data and rule definitions in an operational manner.

9. Note that due to Horn clause theory [SCHÖNING 1989], rules that only allow *one* control action, and not for example the disjunction of control actions, are satisfiable.

Summary

Workflow Definition and Execution

After our overview in Chapter 3 and the ACTIVETFL introduction in Chapter 4, we now describe the AGENTWORK model of *workflow definition* and *workflow execution*. The chapter is organized as follows: Section 5.1 lists the design goals of the AGENTWORK workflow model. Section 5.2 introduces some auxiliary definitions. Section 5.3 describes the workflow definition model which includes activities, control and data flow, and workflow cooperation. Section 5.4 contains the execution model. In Section 5.5 we compare our model with related approaches, namely with petri nets and state/activity charts. A summary and discussion in Section 5.6. concludes the chapter.

Note that only those *temporal* aspects are discussed in this chapter which are necessary to define “normal” workflow execution (e.g., waiting conditions for edges). Temporal aspects that are dedicated to workflow *estimation* – such as how to define and estimate the “duration” of an activity or edge execution – can only be discussed after the entire execution model has been introduced. Furthermore, as the definition of “duration” and the description of workflow estimation takes some place, these aspects are discussed in an own chapter, namely Chapter 6.

5.1 Design Goals

The design goals of the AGENTWORK workflow definition and execution model are as follows:

1. *Expressiveness*

The workflow definition and execution model has to be expressive enough to cover a broad range of business processes. Frequently required control flow structures such as sequences,

conditional branching, and parallel and iterative execution have to be supported. Furthermore, the data flow model has to support object-oriented or object-relational data structures which are necessary to cover the data complexity of domains such as medicine. Inter-workflow dependencies have to be expressed as well.

2. *Adaptation-Oriented Workflow Structuring*

As workflow definitions can become very complex, structuring mechanisms are required that support workflow adaptation. In particular, the control flow and the data flow should be represented explicitly so that they do not have to be reconstructed from an implicit representation (such as ECA rules). Furthermore, nesting structures such as super-workflow/sub-workflow hierarchies should be supported as they facilitate workflow analysis such as temporal workflow estimation.

3. *Formal Foundation*

As AGENTWORK addresses the *semi-automated* adaptation of workflows, a non-ambiguous formal representation of workflows is required. In particular, formal integrity constraints such as that the input of an activity has to be provided entirely by data flow edges are needed to first ensure correct workflow definitions and second that adaptations do not lead to incorrect workflows.

4. *Adaptation-Oriented Execution Support*

The workflow model has to define a clear operational semantics specifying how workflow definitions are executed. This is important as control flow failures affect *currently executed* workflows. In particular, the different states of a workflow and its activities during execution have to be precisely defined. As events inducing control flow failures occur to *cases* or *resources*, it also has to be clearly defined how such cases and resources are assigned to running workflows.

5. *Readability*

Workflow definitions have to be readable not only for computer scientists, but also for users (e.g., physicians). This helps to avoid workflow definitions that may fulfill all integrity constraints but still contain inconsistencies from the application point of view. As an adaptation may require user interaction or even a manual adaptation of some workflow parts, readability also helps to avoid workflow adaptations leading to semantically inconsistent workflows.

Obviously, the listed goals show some dependencies or have a contrasting nature. For example, an increased expressiveness (goal 1) often means decreased readability (goal 5). Therefore, a compromise between such contrasting goals has to be found. For a broader discussion on workflow model requirements including workflow reusability and visualization support we refer to [REICHERT ET AL. 2000, WESKE 2000 C].

To achieve the goals 1-5 listed above, AGENTWORK uses a workflow model based on symmetrical and hierarchical control flow blocks as introduced in [REICHERT & DADAM 1998]. This model will be described in the following.

5.2 Auxiliary Definitions

We start by introducing two auxiliary definitions, namely *named object patterns* and *time-constrained object patterns*. They extend the definitions of object patterns given in Section 4.2.1.5 for the purpose of workflow definition. In the following, *Class* always denotes an arbitrary F-Logic class needed for a particular workflow application (e.g., *Class* may be any class shown in Figure 4-2).

5.2.1 Named Object Patterns

Let P be some object pattern according to 4.2.1.5. A *named* object pattern additionally gives an object matching a pattern a *name*. For example,

$$h: \text{Hemato-Finding}[\text{parameter} = \text{Leukocyte-Count}, \text{unit} = \#/\text{mm}^3] \quad (i)$$

gives an object matching (i) the name h . The *type* of a named object definition is denoted with *Named-Obj-Patt*<*Class*>. For example, the named object definition in (i) is of type *Named-Obj-Patt*<*Hemato-Finding*>.

5.2.2 Time-Constrained Object Patterns

Often it is necessary also to specify time constraints stating for example that an object may not be older than two days w.r.t. the current point in time *now*. This can be done by so-called *time-constrained* object patterns. Formally, if T is a (gregorian) set of points in time according to 4.3.1, a time-constrained object pattern has the form

$$\text{obj-patt NOT-OLDER-THAN } d \quad (ii)$$

where *obj-patt* is a (named) object pattern and d a duration according to 4.3.2. An object o matches a time-constrained object pattern (ii) if it fulfills *obj-patt* and if for the point in time t at which it has been updated at last it holds $\text{now} - d \leq t \leq \text{now}$. An example is

$$h: \text{Hemato-Finding}[\text{parameter} = \text{Leukocyte-Count}] \text{ NOT-OLDER-THAN } (2, \text{day}) \quad (iii)$$

which specifies a *Hemato-Finding* object h where the measured parameter is the leukocyte count and that must not be older than 2 days. For example, such time-constrained patterns can be used to specify the currentness of input data needed by a workflow activity. In this context, they also will play an important role for data flow adaptation. For example, when a new node is added to a workflow and data has to be provided for its input, time-constrained patterns form the basis to decide which data being already available in a workflow may be used for this.

The time-constrained (named) object pattern *type* is denoted with *Time-Constr-Obj-Patt*<*Class*> respective *Time-Constr-Named-Obj-Patt*<*Class*>. For example, the time-constrained object definition in (iii) is of type *Time-Constr-Named-Obj-Patt*<*Hemato-Finding*>. If P is of type *Time-Constr-Named-Obj-Patt*<*Class*> we call the d -parameter in (iii) the time constraint of P .

5.3 Workflow Definition Model

We now describe how workflows are defined. First, we explain how basic activities, resources, and conditions are defined (5.3.1-5.3.4). Second, we describe how the control and data flow of a workflow is defined (5.3.5-5.3.7). Third, we explain the definition of complex (i.e., nested) activities and workflows (5.3.8-5.3.9). Fourth, the definition of workflow cooperation is described (5.3.10).

5.3.1 Basic Activity Definitions

A *basic activity definition* is a declarative description of a basic unit of work. Its main characteristic is that it describes what has to be done on a *high semantic level*. In particular, an activity definition is described in terms of the global data schema (Figure 4-2) and thus in the terminology of the workflow user (e.g., the physician). This supports goal 2 (adaptation-oriented workflow structuring) as it relieves the adaptation from technical details such as the programs needed for the activity execution. Furthermore, goal 5 (readability) is supported by semantically rich activity definitions.

Syntactically, a basic activity definition has the form (*Set*<*P*> denotes a set of elements matching a pattern *P*):

```
Basic-Activity-Def {
  name:           String; // unique identifier
  input:          Set<Time-Constr-Named-Obj-Patt<{Document, Event}>>;
  activity:       Obj-Patt<Activity>;
  output:         Set<Named-Obj-Patt<{Document, Event}>>; }
```

An example for a basic activity definition in a table-oriented notation is shown in Table 5-1. This activity definition specifies that the drug ETOPOSID has to be administered with a dosage of 100 mg per square meter body surface. Furthermore, it is specified that two *Hemato-Finding* objects are expected as input and termed as h_1 and h_2 . They have to fulfill the constraints that the measured parameter is the leukocyte respective thrombocyte count. Additionally, both objects may not be older than 2 days. Furthermore, the activity definition in Table 5-1 specifies that a *Chemo-Report* object is produced as output.

“Administer Etoposid”		
input	activity	output
h_1 : Hemato-Finding [parameter = Leukocyte-Count] NOT-OLDER-THAN (2, day) h_2 : Hemato-Finding [parameter = Thrombocyte-Count] NOT-OLDER-THAN (2, day)	Drug-Administration [drug = “ETOPOSID” dosage = 100 unit = mg/m ² type = infusion]	c: Chemo-Report[]

Table 5-1: Activity definition for an ETOPOSID administration.

We make the following remarks:

- For input and output object patterns (entries *input* and *output*) only the classes *Document* and *Event* are used as we assume that all relevant data that are needed for activity execution are either documents (class *Document*) or data describing events that occurred to a case (i.e., objects derived from class *Event*).
- As the *activity* entry is defined as a *pattern*, not every attribute of the used *Activity* class in *Obj-Patt<Activity>* has to be set to a concrete value at workflow definition time as it has been done in Table 5-1. Rather, attribute values may be specified at execution time by a user assigned to this activity definition (see 5.3.3.1). For example, for the activity definition in Table 5-1 the value of the attribute *dosage* may be left unspecified and then determined by a physician when an activity based on this activity definition is scheduled for execution.
- The assignment of the *Case* instance (*activity-for* association in Figure 4-2) respective of *Resource* instances (*needs-resources* association in Figure 4-2) to the *activity* entry is a matter of workflow execution and will be described in Section 5.4.
- Furthermore, *activity* may refer to objects in *input* or to paths starting from these objects. By this it can be specified how the input objects are used by the activity. For example, if *xrr* is an input object of class *X-Ray-Report* then the *activity* entry for a computer tomography (ct) examination may have the form

CT-Examination[*focus* = *xrr.reported-activity.focus*]

to express that the ct examination shall focus on the focus region of the x-ray examination *xrr* (with *focus* being an attribute of a radiological activity representing the focus area such as the lung; see Figure 4-2). This makes sense as a ct usually is initiated by a preceding x-ray examination with a pathological finding that has to be examined further.

- Note that AGENTWORK activity definitions differ from other workflow approaches w.r.t. the semantic level and the explicitness of what workflow activities do in terms of the application domain. As AGENTWORK has to automatically decide whether an event implies that a currently executed workflow is not adequate anymore, we need a declarative formal description stating what a workflow is currently doing or what it will do in the future. For example, if the situation occurs that a patient must not get the drug ETOPOSID anymore for several days, AGENTWORK has to inspect which workflow activities of running workflows deal with this drug. This is Supported by activity definitions as shown in Table 5-1. Most commercial workflow systems and research prototypes (see Chapter 2) encode this only in a very implicit manner by listing the application programs which are assigned to the workflow activities or by un- or semi-structured text descriptions. For an automated handling this is not sufficient.

With *Basic-Activity-Defs* we denote the basic activity definitions needed for a workflow application. In the following, a basic activity definition such as the one in Table 5-1 will be assigned to a workflow node to describe what is done when the node is executed (see 5.3.5). The

input and output objects are relevant for the definition of the data flow (see 5.3.6).

5.3.2 Activity Compensation

To a basic activity definition A , a compensating activity definition A^{-1} can be assigned. A^{-1} logically undoes or at least minimizes the effects of A . Such compensating activities are needed in case of a workflow rollback. The following points are important to note:

First, the purpose of compensating activities in AGENTWORK is to minimize the effects of activities on the data and communication level, such as cancelling messages that have already been sent. They do *not* contribute to the handling of control flow failures in the sense that they compensate the situation caused by a failure event. It has already been discussed in Chapter 2 (*Related Work*) that workflow compensation models are not adequate for handling control flow failures as defined in this thesis.

Second, for many activities a compensating activity will only be manual or will simply *not exist*. This holds especially for medical domains. For example, for the „Administer ETOPOSID“ activity in 5.3.1 neither the infused drug fluid can be made undone nor would it be appropriate to undo any database operation related to the drug documentation as the drug administration must be documented for legal reasons. If a compensating activity does not exist for an activity definition A , we say that A^{-1} is the so-called NULL activity.

5.3.3 Resource Definitions

Resource definitions specify the resources (i.e., users, programs, and equipment pieces) needed to execute activities. A declarative definition of resources is needed as events occurring to resources (e.g., a computer tomograph may get broken) may cause control flow failures (see 3.3.1).

5.3.3.1 User Definitions

User definitions specify the “profile” of users involved in the execution of an activity. A user definition has the following form:

```
User-Def {
    name:      String; // unique identifier
    user:      Obj-Patt<Staff-Member>;
    proxy:     Obj-Patt<Staff-Member>;}
```

An example is

```
User-Def {
    name:      "Senior-Oncologist";
    user:      Physician[degree = Senior; speciality = "Oncology"];
    proxy:     Physician[degree = Assistant; speciality = "Oncology"];}
```

With *User-Defs* we denote the user definitions needed for a workflow application. The mapping $AUM (\underline{Activity} \rightarrow \underline{User\ Mapping})$

$$AUM: \text{Basic-Activity-Defs} \rightarrow \wp(\text{User-Defs})$$

maps a basic activity definition to a set of user definitions (with \wp being the power set operator). This means that an activity based on a basic activity definition A must be executed by a set of *Staff-Member* objects matching the user definitions in $AUM(A)$. The image of AUM is a *set* as often several users together execute one activity. For example, an x-ray examination usually is executed by a physician and a nurse. If $AUM(A)$ is the empty set then A is said to be an *automated* activity. In this case, at least one program has to be assigned to A (see 5.3.3.2).

5.3.3.2 Program Definitions

Program definitions specify the application programs involved in the execution of an activity. A program definition has the following form:

```

Program-Def {
    name:                String; // unique identifier
    program:              Obj-Patt<Application-Program>;
    controls:             String; // optional; name of controlled equipment (see 5.3.3.3)

```

Program definitions totally abstract from the question where a program is physically installed and executed. This is considered as a matter of some application-integrating middleware such as CORBA. This middleware registers all application programs in a network, provides logical identifiers (as done for example by the CORBA trading service), and determines at execution time at which host which application program code shall be invoked. Furthermore, we do not consider the input and output objects of programs as they are not relevant for workflow adaptation. An example for a program definition is

```

Program-Def {
    name:                "CT-Controller";
    program:              Application-Program[function = "CT-CONTROL"; release = 7.4];
    controls:             "CT"

```

defining the controller program of a computer tomograph. With *Program-Defs* we denote the set of program definitions needed for a workflow application. The mapping $APM (\underline{Activity} \rightarrow \underline{Program\ Mapping})$

$$APM: \text{Basic-Activity-Defs} \rightarrow \wp(\text{Program-Defs})$$

maps a basic activity definition to a set of program definitions. The image of APM is a *set* as often several programs together are involved in the execution of *one* activity. If $APM(A)$ is the empty set

then A is said to be a pure *manual* activity. In this case, at least one user definition according to 5.3.3.1 has to be assigned to A .

5.3.3.3 Equipment Definitions

Equipment definitions specify the *application equipment* needed for the execution of an activity. A typical example for an application equipment is a computer tomograph in a hospital or a copy machine in an office. An equipment definition has the following form:

```
Equipment-Def {
  name:           String;
  equipment:      Obj-Patt<Equipment>;
  controlled-by:  String; }
```

An example is

```
Equipment-Def {
  name:           "CT";
  equipment:      Computer-Tomograph[];
  controlled-by:  "CT-Controller"; }
```

defining a computer tomograph. Analogously to the situation with programs, we denote with *Equipment-Defs* the set of equipment definitions needed for a workflow application. The mapping AEM ($\underline{Activity} \rightarrow \underline{Equipment} \underline{Mapping}$)

$$AEM: \text{Basic-Activity-Defs} \rightarrow \wp(\text{Equipment-Defs})$$

maps a basic activity definition to a set of equipment definitions.

5.3.4 Condition Definitions

For control flow definition purposes, AGENTWORK provides two condition types. A *branching condition* specifies the condition that has to be met to execute a particular set of activities. A *waiting condition* is used to define the temporal distance between two activities.

A *branching condition* has the following form:

```
Branching-Cond-Def {
  name:           String; // optional
  input:          Set<Time-Constr-Named-Obj-Patt<{Document, Event}>>;
  condition:      Formula; }
```

where *condition* is a formula according to 4.2.3 which is defined on objects in *input*. An example is:

```
Branching-Cond-Def {
  name:          "Severe-Leukocyte-Finding";
  input:         {h: Hemato-Finding[parameter = Leukocyte-Count]};
  condition:     h.value < 1000 AND h.unit = #/mm3; }
```

A *waiting* condition has the form:

```
Waiting-Cond-Def {
  name:          String; // optional
  min:           Duration;
  max:           Duration; }
```

with *Duration* being the temporal distance type as defined in 4.3.2. An example is:

```
Waiting-Cond-Def {
  name:          "Chemo-Therapy-Recovery";
  min:           (1, week);
  max:           (2, week); }
```

that may be inserted between two chemotherapy activities to specify that the patient should recover from the first chemotherapy activity at least one week but not longer than two weeks. If *min* = *max*, this specifies an exact waiting time.

The precise semantics of these conditions is explained in 5.3.5.1 where they are assigned to control flow edges. With *Branching-Cond-Defs* and *Waiting-Cond-Defs*, we denote the sets of definitions for branching and waiting conditions needed for a workflow application.

5.3.5 Control Flow Definitions

We now describe the control flow model of AGENTWORK. We first introduce the definition of the so-called *basic* control flow (5.3.5.1). Second, we introduce several control flow constraints that have to be met to facilitate workflow adaptation without reducing expressiveness too much (5.3.5.3). In particular, these constraints enforce that the control flow model is organized by symmetrical blocks.

5.3.5.1 Basic Control Flow Definitions

A *basic* (or unnested) *control flow* definition *CF* over *Basic-Activity-Defs* is a tuple

$$CF = (Activity-Nodes, NAM_{basic}, Control-Nodes, Edges, BC, WC)$$

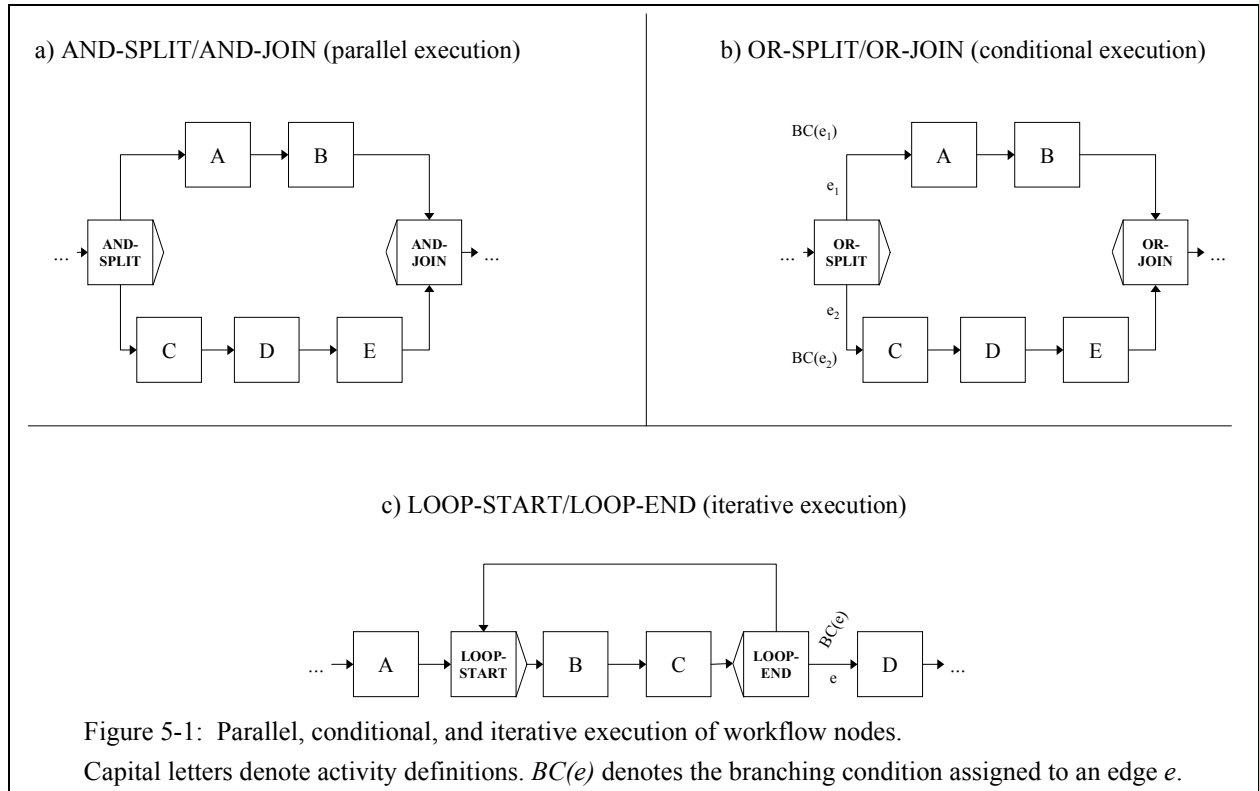
with the following meaning:

- *Activity-Nodes* is the set of *activity* nodes.
- The function

$$NAM_{basic} : Activity-Nodes \rightarrow Basic-Activity-Defs$$

maps a basic activity definition to every activity node ($NAM_{basic} = \underline{Node} \rightarrow \underline{Activity\ Definition\ Mapping}$). We call n an *A-node*, if $NAM_{basic}(n) = A$, i.e., if n executes an activity based on the activity definition A .

- *Control-Nodes* is the set of *control flow* nodes (with $Control-Nodes \cap Activity-Nodes = \emptyset$). Figure 5-1 and Table 5-2 show the control node types supported by AGENTWORK. Note that the relationship between control *nodes* and control *actions* is that control nodes specify the control flow (i.e., conditional, parallel, or iterative activity execution) which then may be adapted due to triggered control actions.
- *Edges* $\subset (Activity-Nodes \cup Control-Nodes) \times (Activity-Nodes \cup Control-Nodes)$ is the set of directed control flow edges. For an edge $e = (n, m) \in Edges$, n is called *source node* and m is



Node Type	Explanation
START END	Specify the beginning and the end of a workflow, i.e., a START node may not have a predecessor node and an END node may not have a successor node.
AND-SPLIT AND-JOIN	Specify the beginning and end of a <i>parallel</i> execution (Figure 5-1 a). A necessary condition for continuing the workflow after the AND-JOIN node is that <i>all</i> paths have been executed <i>entirely</i> .
OR-SPLIT OR-JOIN	Specify the beginning and end of a <i>conditional</i> execution (Figure 5-1 b). The question whether a conditional execution has an "exclusive or" semantics (exactly one from many) or not is a matter of the condition definitions assigned to it. A necessary condition for continuing the workflow after the OR-JOIN node is that <i>all</i> paths that qualified for execution have been executed <i>entirely</i> .
LOOP-START LOOP-END	Specify the beginning and the end of a loop with REPEAT-UNTIL semantics, i.e., the loop sequence is executed at least once and as long as the termination condition does not hold (e.g., $BC(e)$ in Figure 5-1 c). Loops with a WHILE-DO or FOR-TO semantics are not explicitly supported, but can easily be specified either by placing a conditional branching before the LOOP-START node (WHILE-DO) or by embedding the loop sequence within some "counting" activities and by specifying an appropriate termination condition (FOR-TO).
Definitions:	Nodes of type START, AND-SPLIT, OR-SPLIT or LOOP-START are called "opening" nodes (as they open a workflow, a parallel execution etc.). Nodes of type END, AND-JOIN, OR-JOIN or LOOP-END are called "closing" nodes.

Table 5-2: Control node types.

called *target node*. A control flow edge $e = (n, m)$ has the semantics that a necessary precondition for executing m is that the execution of n has been completed successfully.

- *BC: Edges \rightarrow Branching-Cond-Defs* assigns branching conditions (see 5.3.4) to edges. By convention, we set $BC(e) := \text{NULL}$ (the "null condition" which is always TRUE) if no branching condition shall be assigned to e . For an edge $e = (n, m)$ with $BC(e) \neq \text{NULL}$ it holds that a necessary precondition to execute m is that $BC(e)$ is fulfilled.
To avoid deadlocks, such branching conditions will be restricted to edges having OR-SPLIT or LOOP-END nodes as source (see 5.3.5.3).
- *WC: Edges \rightarrow Waiting-Cond-Defs* assigns waiting conditions (see 5.3.4) to edges. By convention, we set $WC(e) := \text{NULL}$ (the "null condition" which is always TRUE) if no waiting condition shall be assigned to e . For an edge $e = (n, m)$ with $WC(e) \neq \text{NULL}$ it holds that a necessary precondition to execute m is that at least the time specified by the *min* entry of $WC(e)$ has elapsed and that the time specified by the *max* entry of $WC(e)$ has not yet elapsed (since n has been executed).

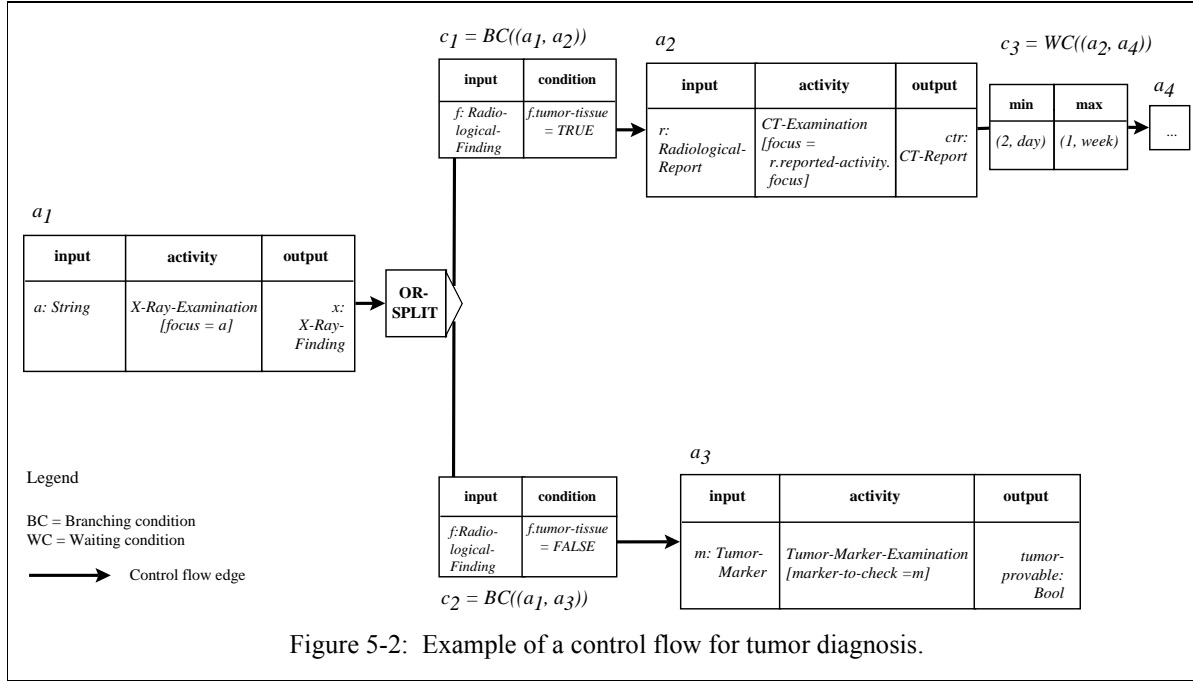


Figure 5-2 gives an example for a control flow definition of the HEMATOWORK system. This control flow starts with an x-ray examination (activity node a_1). If the x-ray examination detects tumor tissue, this tissue is further inspected by a computer tomography examination (a_2). If no tumor tissue is detected by the x-ray examination, a controlling tumor marker examination is executed (a_3) to check whether there is tumor issue not detectable by radiological examinations.

5.3.5.2 Control Flow Paths, Predecessor and Successor Nodes and Edges

Let $CF = (Activity\text{-}Nodes, NAM_{basic}, Control\text{-}Nodes, Edges, BC, WC)$ be a basic control flow definition. Then, a *control flow path* is any sequence

$$(n_1, n_2), (n_2, n_3), (n_3, n_4) \dots (n_{k-1}, n_k) \in Edges^{k-1}.$$

Alternatively, we may also write $n_1 \rightarrow n_2 \rightarrow n_3 \rightarrow \dots \rightarrow n_k$ for such a control flow path.

If n and m are nodes or edges of CF , then n is called *predecessor* node or edge of m if it exists at least one control flow path leading from n to m . Analogously, n is called *successor* node or edge of the node or edge m if it exists at least one control flow path leading from m to n . The node n is called *direct predecessor* or *successor* node of m , if the control flow path between n and m does not contain any other node than n and m .

5.3.5.3 Control Flow Constraints

So far, the definition of Section 5.3.5.1 allows to specify arbitrary control flows. In particular, it allows to insert control flow edges between arbitrary activity and control nodes and to assign branching or waiting conditions to any of these edges. However, such arbitrary control flows significantly increase the risk of deadlocks or node starvation during execution and complicate temporal workflow estimation. Thus, it is necessary to define some workflow constraints so that workflow definitions remain controllable and readable without restricting them too much. For this purpose, we list three constraints which forbid 1. isolated nodes, 2. activity splits and joins, and 3. unbalanced control nodes. These three constraints correspond to the workflow structuring rules introduced in [REICHERT 2000]. For the purposes of this thesis, we give a semi-formal description of these constraints. A formal definition based on set-theory can be found in chapter 3 of [REICHERT 2000].

Control Flow Constraint 1 (Node Reachability)

For every node $n \in \text{Activity-Nodes} \cup \text{Control-Nodes}$ it must hold that n has to be *reachable* from the START node, and that the END-node is reachable from n . This means that there must be at least one control flow path $\text{START} \rightarrow n_1 \rightarrow n_2 \dots \rightarrow n_k \rightarrow n \in \text{Edges}^{k+1}$, and at least one control flow path $n \rightarrow n_1 \rightarrow n_2 \dots \rightarrow n_k \rightarrow \text{END} \in \text{Edges}^{j+1}$.

By this constraint, isolated nodes such as the *B*-node in Figure 5-3 are avoided.

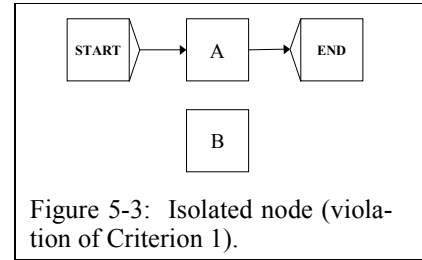


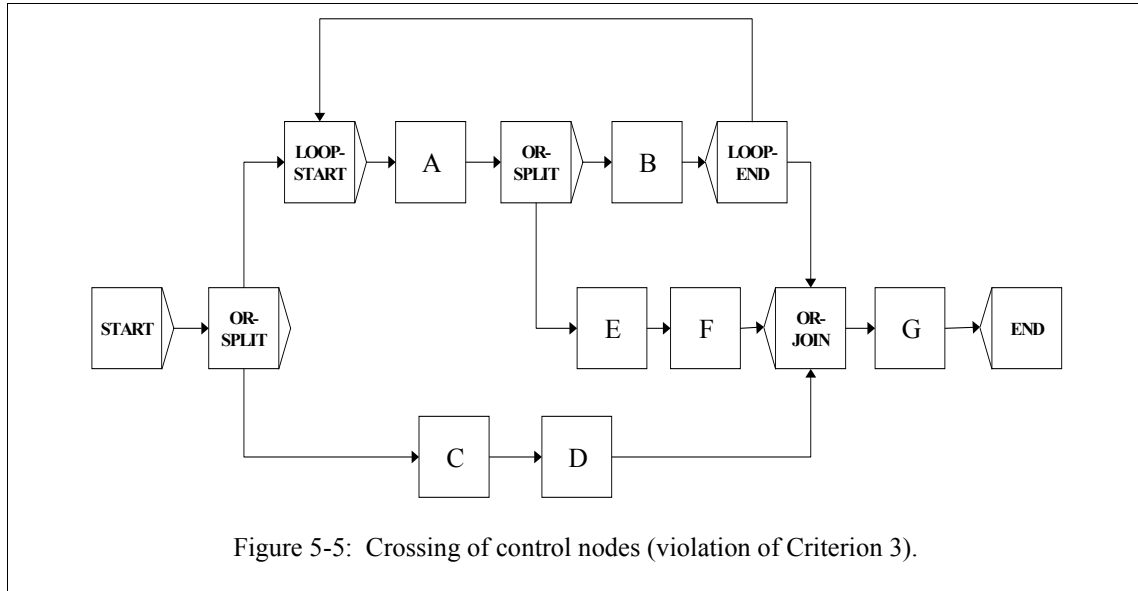
Figure 5-3: Isolated node (violation of Criterion 1).

Control Flow Constraint 2 (No Activity Split/Join)

Only AND-SPLIT, OR-SPLIT and LOOP-END nodes may have more than one outgoing edge. In particular, the edges with a branching condition are exactly those edges having an OR-SPLIT or LOOP-END as source, i.e., for $e = (n, m)$ with $BC(e) \neq \text{NULL}$, n must be an OR-SPLIT or LOOP-END node. A LOOP-END node must have exactly two outgoing edges, one for the next loop iteration, and one for loop termination. To the latter edge, a branching condition has to be assigned playing the role of a loop termination condition (see Figure 5-1 c).

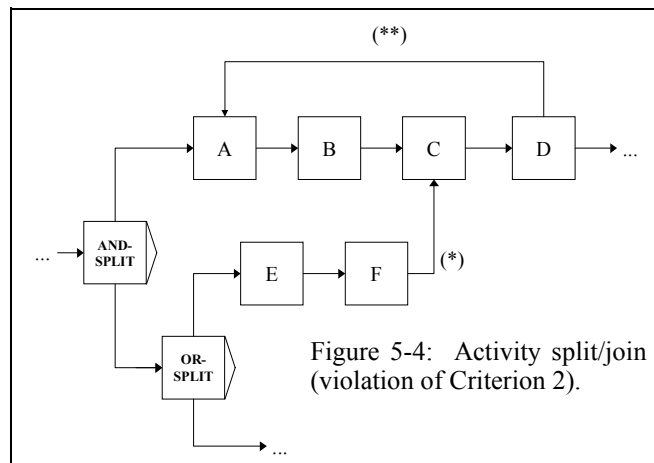
Analogously, only AND-JOIN, OR-JOIN and LOOP-START nodes may have more than one incoming edge. A LOOP-START node must have exactly two incoming edges, one from the node executed last before the first loop iteration and one from the LOOP-END node (see Figure 5-1 c).

By this constraint, implicit and unclear split, join and loop semantics that are not under the control of AND-SPLIT/AND-JOIN, OR-SPLIT/OR-JOIN and LOOP-START/LOOP-END nodes are avoided. For example, in Figure 5-4 on one side it would be unclear for the control flow edge from the *F*-node to the *C*-node (*) what shall happen if the *F*-node is not executed at all in case the path $\text{OR-SPLIT} \rightarrow E \rightarrow F$ did not qualify for execution (as a necessary condition to execute the *C*-node is that the *F*-node has been executed successfully). On the other side, if the *F*-node would be exe-



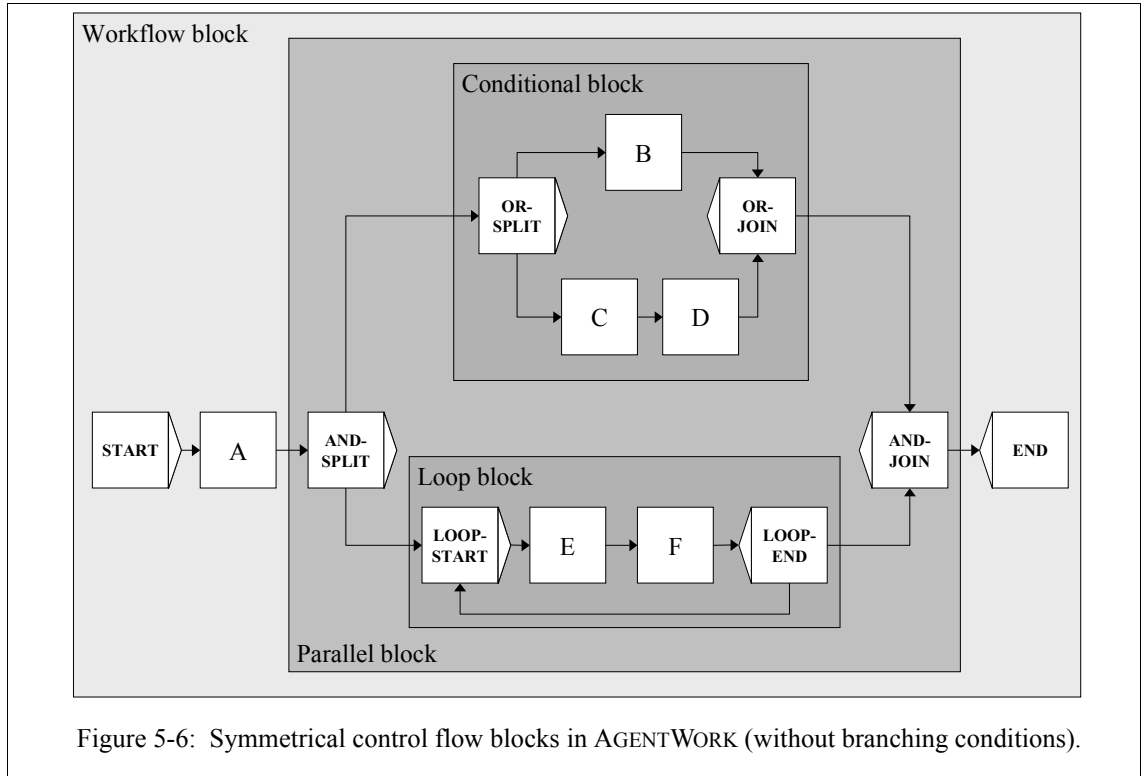
cuted it would be unclear for which iteration (e.g., the first?) of the implicit loop (**) it would hold that the *C*-node must not be executed before the *F*-node has been completed successfully.

More important, the estimation of a workflow's duration is facilitated if activity cross-overs such as (*) and implicit loops such as (**) are avoided. Otherwise, a control flow path may have arbitrary control flow dependencies with other paths which make it more difficult to estimate the path's duration. For example, in Figure 5-4 the duration of the path $A \rightarrow B \rightarrow C \rightarrow D$ also depends on the duration of the conditional path $OR-SPLIT \rightarrow E \rightarrow F$, as *C* cannot be executed before *F* has not been completed. Thus, if such arbitrary dependencies are avoided workflow estimation durations become more precise.



In particular, constraints 1 and 2 together imply that an activity node always has exactly one incoming and one outgoing control flow edge. This characteristic will be frequently used in the following for workflow execution and workflow analysis.

However, constraints 1 and 2 do not forbid “pathological” crossings between control nodes such as the one shown in Figure 5-5 where one path starting at an *OR-SPLIT* “jumps out” of a loop. As



such crossings reduce readability and make temporal estimations much more difficult and thus are incompatible with goals 5 and 2, they should not be allowed. Therefore, we introduce another constraint which restricts control flow definitions to so-called *symmetrical control flow blocks*. This structuring principle is known from structured programming [DIJKSTRA 1968, HERRTWICH & HOMMEL 1994] and has recently also been applied to workflow management [REICHERT & DADAM 1998, KIEPUSZEWSKI ET AL. 2000].

Control Flow Constraint 3 (Symmetrical Blocks)

For an AND-SPLIT or OR-SPLIT node there has to be a one-to-one corresponding AND-JOIN respective OR-JOIN node joining all paths starting at the AND-SPLIT respective OR-SPLIT node. In particular, the number of outgoing edges of the AND-SPLIT respective OR-SPLIT node must be the same as the number of incoming edges at the corresponding AND-JOIN respective OR-JOIN node. In Figure 5-5, this constraint is violated as the paths starting at the two OR-SPLIT nodes are joined by only one OR-JOIN node.

Analogously, for every LOOP-START node n there must be exactly one LOOP-END node m with an edge (n, m) .

A workflow part starting from an AND-SPLIT, OR-SPLIT, or LOOP-START node and closed by

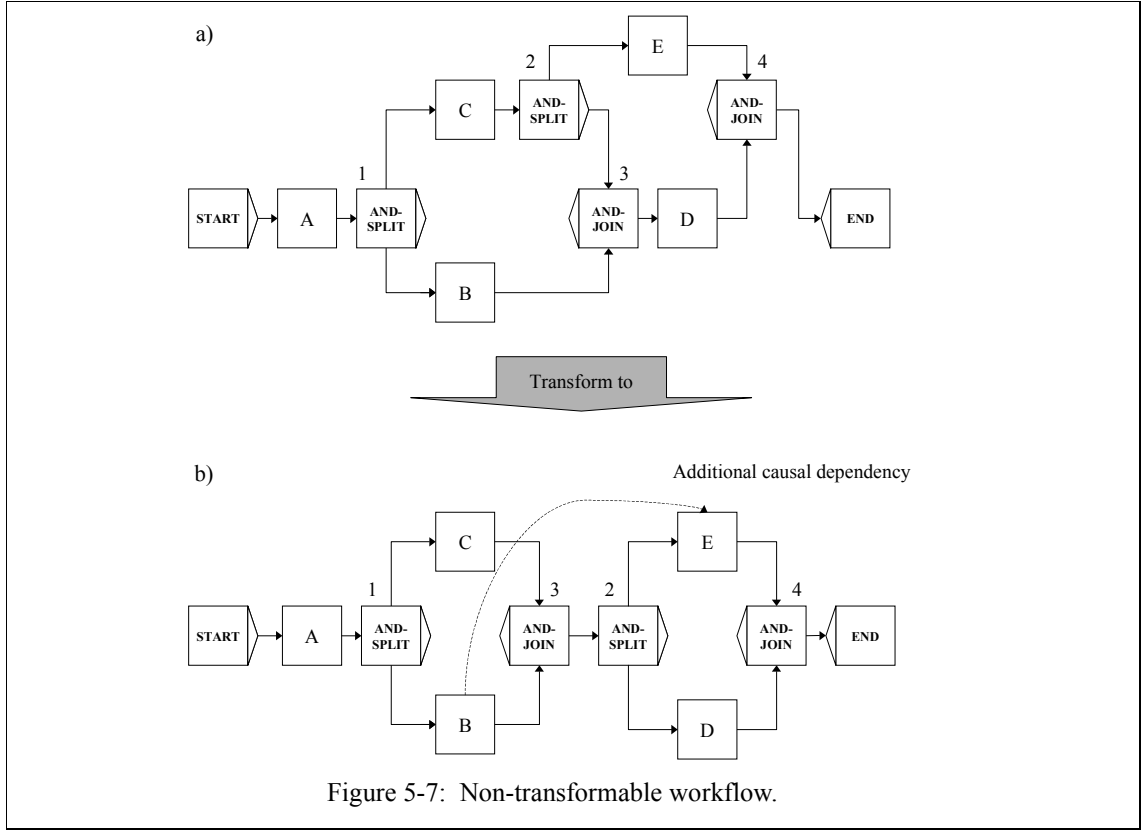
the corresponding AND-JOIN, OR-JOIN, respective LOOP-END node is called parallel, conditional, respective loop *block* (Figure 5-6). The whole workflow starting from the START node and ending at the END-node is also called *workflow block*.

We make the following remarks:

- Constraint 3 allows that blocks may be arbitrarily nested. This means that instead of consisting solely of activity nodes a block may also contain other blocks. Together with constraint 2, constraint 3 enforces that such a sub-block entirely belong to its super-block, i.e., that it cannot not “share” any of its nodes with other super-blocks (i.e., the relationship between blocks and sub-blocks is a one-to-many relationship). Figure 5-6 shows a parallel block containing a conditional and a loop block.
- *Waiting* conditions can be assigned to *all* control edges. If both a branching *and* a waiting condition have been assigned to an edge $e = (n,m)$ (i.e., $BC(e) \neq \text{NULL}$ and $WC(e) \neq \text{NULL}$), this means by default that $BC(e)$ must hold *at the end* of the waiting time. If the workflow modeler needs the semantics that $BC(e)$ must hold *at the beginning* of the waiting time this has to be specified explicitly.
- Furthermore, for an OR-SPLIT node and its branching conditions it must be guaranteed that *at least one* path will be executed.
- Obviously, especially constraints 2 and 3 reduce workflow expressiveness for the sake of readability and analysis. As shown in [KIEPUSZEWSKI ET AL. 2000], not every workflow can be transformed into a workflow meeting the constraints listed above. For example, the workflow in Figure 5-7 a) violates constraint 3 (as the path $C \rightarrow \text{AND-SPLIT} \rightarrow E$ is not leading to the AND-JOIN node 3, if we assume that 1 corresponds to 3 and 2 to 4¹). At first glance, it seems that the workflow Figure 5-7 b) which meets constraints 1-3 is equivalent. However, it introduces an additional causal dependency between B and E , as now E cannot be executed before B has been completed. This was not expressed in workflow a). The alternative to use only one AND-SPLIT/AND-JOIN block with two parallel paths $C \rightarrow E$ and $B \rightarrow D$ also would not solve the problem as then the dependency between C and D would be lost.

In Section 5.4, after having described the different *states* a node may have during execution, we will use so-called *synchronization edges* [REICHERT & DADAM 1998] to synchronize nodes belonging to different parallel control flow paths within an AND-SPLIT/AND-JOIN block or an OR-SPLIT/OR-JOIN block. This additional element allows to express workflows such as the one shown in Figure 5-7 a) but does not violate constraints 1-3 and not leading to deadlocks or unclear execution semantics.

1. If we assume that 1 corresponds to 4 and 2 to 3, then constraint 3 would also be violated as the path $\text{AND-SPLIT} \rightarrow E$ would not lead to node 3.



5.3.5.4 Minimal Block

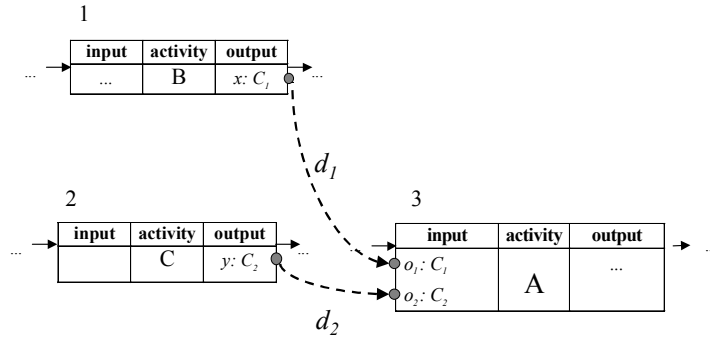
We conclude this section on control flow specification with a definition of the so-called *minimal block of a node set*. This definition will frequently be used especially for workflow estimation purposes and control flow adaptations.

Let S be an arbitrary set of nodes of a workflow control flow definition. The minimal block MB_S is that AND-SPLIT/AND-JOIN, OR-SPLIT/OR-JOIN, LOOP-START/LOOP-END, or START/END block which fulfills the following two conditions:

1. S is entirely contained in MB_S (with $S \neq MB_S$, i.e., MB_S contains at least one node not belonging to S).
2. There is no other block $MB'_S \neq MB_S$ that contains S (with $S \neq MB'_S$) and that itself is entirely contained in MB_S .

Figure 5-8: Data flow edges.

The non-dashed lines denote the control flow.



For example, the minimal block of the *B*-node in Figure 5-6 is the OR-SPLIT/OR-JOIN block while the minimal block of the *A*-node is the whole workflow. By postulating $S \neq MB_S$ in condition 1, we enforce that the minimal block of a minimal block *MB* is not *MB* itself, but the next surrounding block (this has only technical reasons for workflow estimation and adaptation). An algorithm for determining minimal blocks can be found in chapter 3 of [REICHERT 2000].

Note that in contrast to [REICHERT 2000], sequences of activity nodes such as $E \rightarrow F$ in Figure 5-6 are not viewed as entire blocks, i.e., the minimal block always is the surrounding AND-SPLIT/AND-JOIN, OR-SPLIT/OR-JOIN, LOOP-START/LOOP-END, or START/END block. This has only technical reasons w.r.t. workflow estimation and adaptation.

5.3.6 Data Flow Definitions

So far, we have specified which objects are processed by activities or branching conditions. However, we also need to specify where these objects "come from" and where they are "moved to", i.e., how the data shall flow during workflow execution. As AGENTWORK focuses on control flow adaptation and views data flow adaptation "only" as a consequence of control flow adaptation (e.g., for added nodes the needed input objects additionally have to be provided by the data flow), this thesis provides a straightforward data flow model.

The basic data flow construct in AGENTWORK is the so-called *data flow edge*. Data flow edges specify how input objects are "filled" or "initialized" by other objects, or how output objects are written to an object extension that may represent a table in a relational database.

The principal structure of a data flow edge is best illustrated by an example (Figure 5-8): Let C_1 and C_2 be two ACTIVETFL classes, and let us assume that activity node 3 in Figure 5-8 needs the input objects $o_1: C_1$ and $o_2: C_2$. If we want to "fill" o_1 and o_2 by using two objects $x: C_1$ and $y: C_2$ provided as output objects by two other activity nodes 1 and 2, we can specify this by two tuples:

$$d_1 = (1.x, 3.o_1) \quad \text{and} \quad d_2 = (2.y, 3.o_2) \quad (iv)$$

with the meaning that first the output object x of node 1 is mapped to the input object o_1 of node 3,

and that second the output object y of node 2 is mapped to the input object o_2 of node 3. Whether “mapped” means that o_2 becomes a reference to x or that x is copied to o_2 is considered as a matter of implementation. In the current AGENTWORK implementation (see Chapter 11), the mappings by default are implemented by a reference semantics to achieve a high data currentness. Thus, in the following we will read a tuple such as $(1.x, 3.o_2)$ as “ o_2 becomes a reference to x_1 ”.

As the tuples in (iv) can graphically be illustrated as edges connecting input and output objects of different nodes (Figure 5-8), we call them data flow *edges* in the following.

We now describe the data flow between activity nodes, branching conditions, and object extensions in Section 5.3.6.1. Then, we roughly sketch the data flow between activity nodes and programs in Section 5.3.6.2, and motivate why this second data flow type can be neglected in this thesis. For the sake of simplicity, we assume *unique* names for all input and output object definitions used within a control flow definition.²

5.3.6.1 Data Flow between Activity Nodes, Branching Conditions, and Object Extensions

Let CF denote a basic control flow definition according to Section 5.3.5.1. To define the data flow between activity nodes, branching conditions, and object extensions for CF , we introduce the following sets:

Activity-Input denotes the set of *all* activity *input* objects of *all* activity nodes of CF . For example, for the workflow sketched in Figure 5-8, o_1 and o_2 belong to *Activity-Input*. Analogously, *Activity-Output* denotes the set of *all* activity *output* objects of *all* activity nodes of CF . *Cond-Input* denotes the set of *all* *input* objects of *all* control flow edges of CF . For example, the *Hemato-Finding* object h of the “*Severe-Leukocyte-Finding*” condition in Section 5.3.4 would belong to *Cond-Input*, if this condition would be assigned to a control flow edge of CF . Note that *waiting* conditions do not need objects so that we don’t have to consider them for data flow.

With *Obj-Extensions* we denote the set of object extensions according to 4.2.1.8. We recall from Chapter 4 that in AGENTWORK these object extensions are used as abstractions of physical data sources such as file systems or relational databases. In particular, some object extension may be also used for the communication with remote workflow systems, i.e., objects inserted into such an extension may be read by such remote workflow systems. The mapping between these object extensions on one side and the physical data sources or remote workflow systems on the other side is the task of the communication and integration layer (Chapter 11).

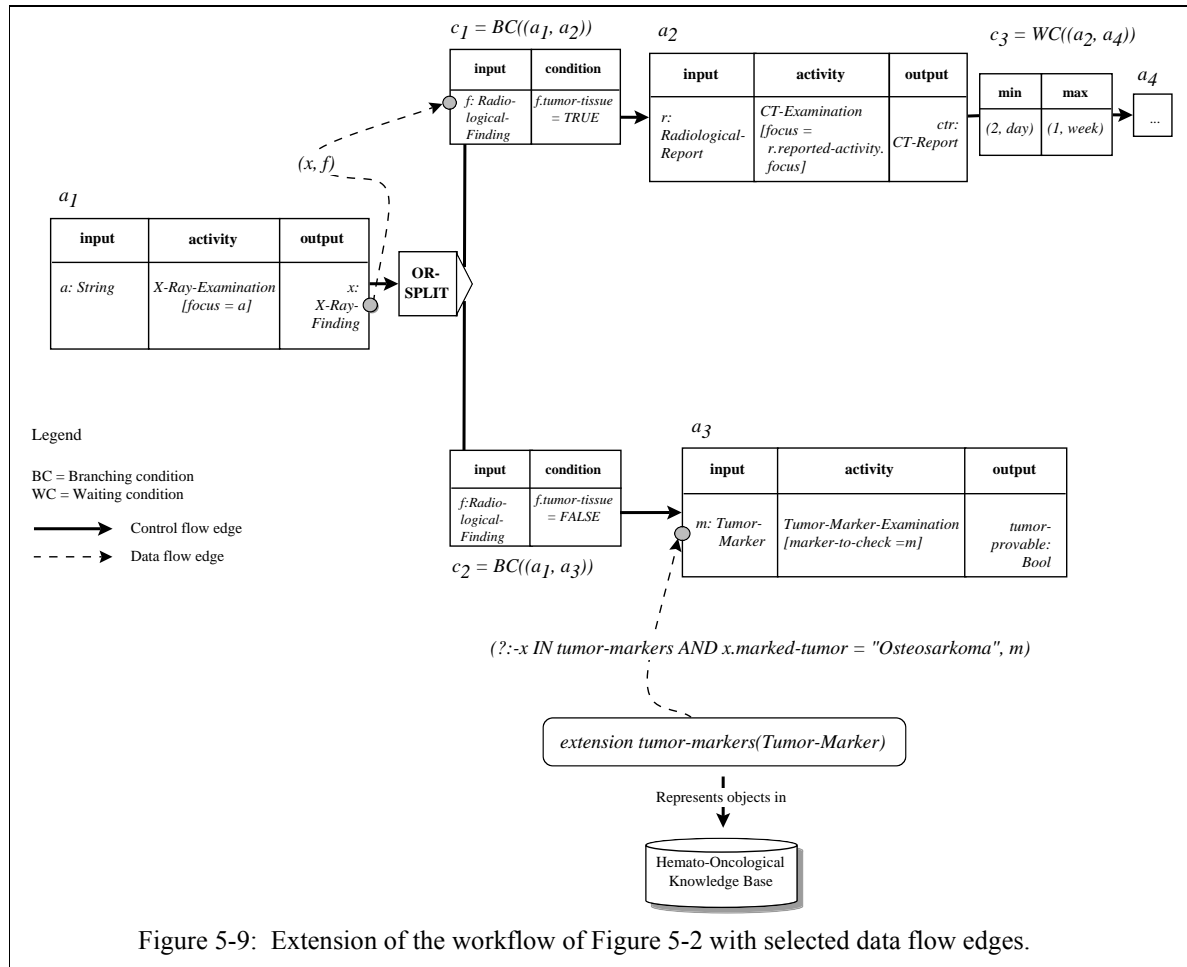
Concerning the data flow between activity nodes, branching conditions, and object extensions, we distinguish two types of data flow edges, namely such for *internal* and *external* data flow. This distinction is motivated by their different handling during workflow execution.

2. This can be enforced, for example, by postulating unique names within a single activity or branching condition, and by indexing these names by the node and edge identifiers.

Internal Data Flow: A tuple (s, t) is called an *internal* data flow edge, if it holds:

- $(s, t) \in$
1. $Activity-Output \times Activity-Input \cup$ // from activity output to activity input
 2. $Activity-Output \times Cond-Input$ // from activity output to branching conditions
- and
- s matches t (according to Section 4.2.1.7)

We call these tuples *internal* data flow edges, as they describe the data flow between “internal” workflow elements such as activities and conditions. An example for such an internal data flow edge is shown in Figure 5-9. In this figure, the internal data flow edge (x, f) between activity node



a_i and the upper branching condition c_i specifies that the *X-Ray-Finding* output object x of a_i shall be mapped to the c_i input object f of class *Radiological-Finding*.

External Data Flow: A tuple (s, t) is called an *external* data flow edge, if it fulfills one of the two definitions:

1. Data flow from object extensions to activity input or condition input objects (“reading“ edge):
 - s is an ACTIVETFL query on an object in *Obj-Extensions*,
 - $t \in \text{Activity-Input} \cup \text{Cond-Input}$
 - the result object of s matches t (according to Section 4.2.1.7).
2. Data flow from activity output to object extensions (“writing“ edge):
 - $s \in \text{Activity-Output}$
 - t is an insert or update operation on an object extension ext and uses s (e.g., inserts s into ext)
 - for the class $Class_s$ of s it holds: $Class_s \text{ IS-A } Class_{ext}$, where $Class_{ext}$ is the class for which the extension ext has been defined (according to Section 4.2.1.8), e.g., $Class_{ext} = \text{Hemato-Finding}$ for $ext = \text{hemato-findings}(\text{Hemato-Finding})$.

We call these tuples *external* data flow edges, as they describe the data flow between workflow elements and object extensions representing “external” data sources. An example for an external data flow edge meeting definition 1 is given in Figure 5-9. There the external data flow edge

$$(?-x \text{ IN tumor-markers AND } x.\text{marked-tumor} = \text{“Osteosarkoma“}, m) \quad (v)$$

specifies that first the marker shall be found in the object extension *tumor-markers* that is able to detect tissue of an osteosarkoma (a bone tumor type). Second, the retrieved object x shall be mapped to the *Tumor-Marker* input object m of node a_3 (assuming that there is exactly one marker for each tumor type).

5.3.6.2 Data Flow between Activity Nodes and Programs

The data flow between activity nodes and the programs assigned to these activity nodes is entirely determined by the respective activity *definitions* and program *definitions*. In particular, the input objects of an activity definition have to be mapped to the objects needed for a program execution, and the objects produced by a program execution have to be mapped to the output objects of an activity definition. The important point is that this mapping does *not* additionally depend on the location of the node which uses a particular activity definition (in contrast to internal and external data flow described above). As a consequence, the adaptation of data flow between activity nodes and programs in case of control flow failures is very simple. For example, if an activity node is dropped, the data flow between its activity definition and the program definition has entirely to be dropped as well. If a node is added, the full mapping between the node’s activity definition and the programs needed to execute the node has to be established. For this reason, we do not consider the

data flow between activity nodes and programs anymore in the context of handling control flow failures.

5.3.6.3 Data Flow Definitions

A workflow data flow definition DF then is a tuple

$$DF = (Internal\text{-}Data\text{-}Flow, External\text{-}Data\text{-}Flow)$$

where *Internal-Data-Flow* is a set of internal data flow edges, and *External-Data-Flow* a set of external data flow edges according to Section 5.3.6.1.

5.3.6.4 Data Flow Constraints

To avoid complications during workflow execution, a data flow definition should meet some correctness constraints. As this thesis does not focus on data flow aspects, we restrict ourselves to two important data flow constraints, namely input completeness and forward-oriented data flow. A more comprehensive discussion on data flow constraints can be found in [REICHERT 2000].

In the following, $input_x$ denotes the set of input objects of an activity node or branching condition x . Analogously, $output_x$ denotes the set of output objects an activity node x is providing as output.

Data Flow Constraint 1 (Input Completeness)

Let x be an activity node or a control flow edge with branching condition. For every element $c \in input_x$, there has to be a data flow edge so that c can be initialized when the control flow reaches x . Formally, it has to hold:

For each $t \in input_x$ it holds:

It exists $(r, s) \in Internal\text{-}Data\text{-}Flow \cup External\text{-}Data\text{-}Flow$ with: $s = t$

Of course, due to conditional branching this constraint not yet guarantees that all input objects can be initialized at execution time. For example, imagine an internal data flow edge where the source node belongs to an OR-SPLIT/OR-JOIN block and where the target node is located behind the resp. OR-JOIN node. Then, if the source node would not be executed as its path has not qualified for execution, the target node may not receive some of its input objects anymore. One possibility to cope with this problem would be to forbid any constellation where the target node of an internal data flow edge may be executed while the resp. source node may not be executed as its path is conditional. However, this is viewed as too restrictive. The better alternative is – as described in 5.4 (*Workflow Execution Model*), in particular 5.4.3.5 – to allow such conditional data flow constellations, and to claim for missing data at execution time, if necessary.

Data Flow Constraint 2 (Forward-Oriented Data Flow)

A data flow edge $e = (s, t) \in Internal\text{-}Data\text{-}Flow$ is not allowed to go “backwards” to a preceding

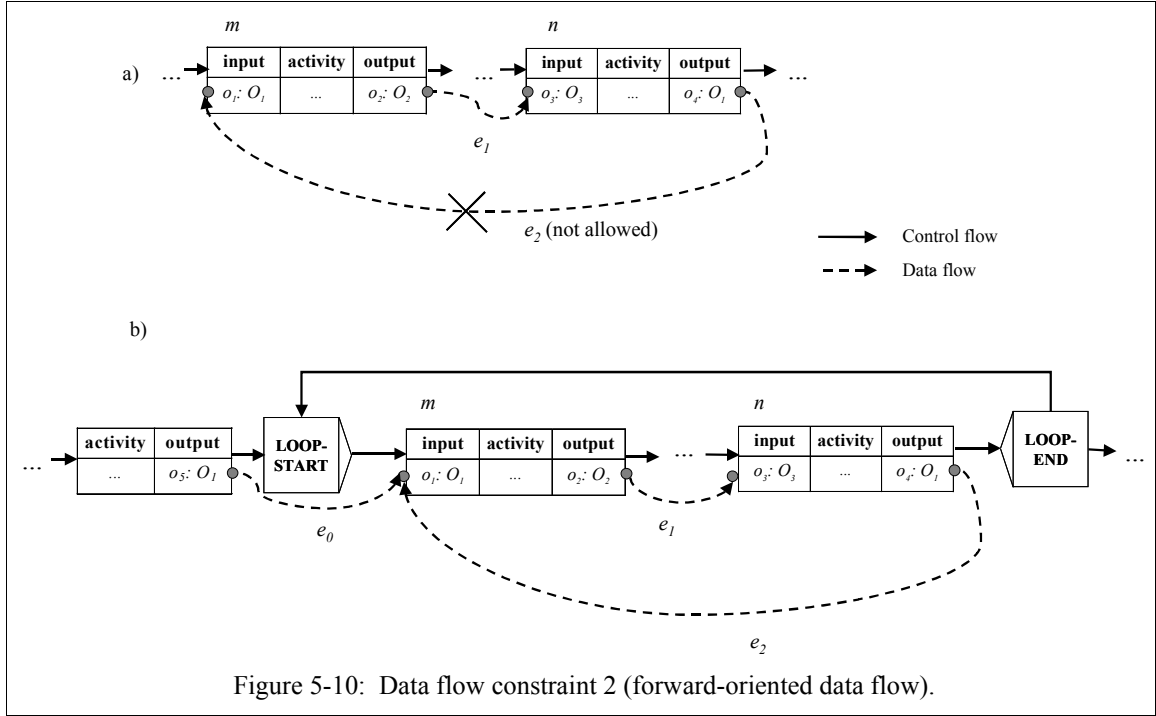


Figure 5-10: Data flow constraint 2 (forward-oriented data flow).

activity node or control flow edge. For example, in Figure 5-10 a) the data flow edge e_2 mapping o_o back to o_i is not allowed as node m cannot receive o_o because n is executed later in the control flow.

The only exception allowing a “backward-oriented” data flow is in the context of loops (Figure 5-10 b). However, this requires that there is an additional data flow edge (e.g., e_0 in Figure 5-10 b) specifying from where the input data shall be retrieved for the first loop iteration.

Formally, a forward-oriented data flow is achieved if for any internal data flow edge $e = (s, t)$ between two nodes x and y there is at least one control flow path from x to y .

5.3.7 Basic Workflow Definitions

A basic (or unnested) workflow definition W over *Basic-Activity-Defs* is a triple

$$W = (\text{name}, CF, DF) \quad \text{where}$$

- *name* is a unique name for the workflow definition,
- *CF* is a basic control flow definition meeting all control flow constraints, and
- *DF* is a data flow definition according to 5.3.6.3 meeting all data flow constraints of 5.3.6.4.

After having introduced basic activities and basic workflow definitions, we now introduce so-called *complex* activity definitions and *complex* workflow definitions to allow sub-workflows within a workflow. Generally, AGENTWORK assumes that any workflow at one time may be executed as a sub-workflow *and* at another time may be executed as a “stand-alone” workflow. Sub-workflows support goal 2 (adaptation-oriented workflow structuring). For instance, if a workflow is used in different workflows as a sub-workflow, its temporal estimation can be re-used. Furthermore, sub-workflows increase workflow readability (goal 5) as they allow to hierarchically decompose workflows.

5.3.8 Complex Activity Definitions

A *complex activity definition* is used as a placeholder for a sub-workflow and describes the input and output data of this sub-workflow. It has the following form:

```
Complex-Activity-Def {
  name:                String;
  input:                Set<Time-Constr-Named-Obj-Patt<Object>>;
  sub-workflow:         String;
  output:               Set<Named-Obj-Patt<Object>>; }
```

where *sub-workflow* is the name either of a *basic* workflow definition according to 5.3.7 or of a *complex* workflow definition introduced below (see 5.3.9). At workflow execution time, the workflow identified by *sub-workflow* has to be executed when the control flow reaches a node *n* to which a complex activity definition has been assigned. This execution has to be performed *synchronously*, i.e., the control flow after *n* can only continue after the whole sub-workflow has been executed.

Concerning data flow aspects, the mapping

1. from the input objects of a complex activity definition to the input objects of its sub-workflow’s activity nodes and control flow edges, and
2. from the activity output objects of the sub-workflow to the output objects of the complex activity definition

is defined analogously to the internal data flow described in 5.3.6.1 so that we omit details here.

With *Complex-Activity-Defs* we denote the set of all complex activity definitions with data flow needed for a workflow application.

5.3.9 Complex Workflow Definitions

We can now give the full definition of a workflow with control flow, data flow and sub-workflows. Let therefore $Activity-Defs = Basic-Activity-Defs \cup Complex-Activity-Defs$ be the union of all basic

and complex workflow activity definitions. A *complex* (or *nested*) workflow definition $W_{complex}$ then is a triple

$$W_{complex} = (name, CF_{complex}, DF) \quad \text{where}$$

- $name$ is a unique name for the workflow definition
- $CF_{complex}$ is a control flow as defined in 5.3.6.3 but where the node activity mapping

$$NAM_{basic} : Activity-Nodes \rightarrow Basic-Activity-Defs$$

has been extended to the mapping

$$NAM : Activity-Nodes \rightarrow Activity-Defs$$

(so that also a *complex* activity definition can be assigned to an activity node).

- DF is a data flow definition according to 5.3.6.

In the sequel, an activity node n with $NAM(n) \in Basic-Activity-Defs$ is called a *basic* activity node. Analogously, an activity node with $NAM(n) \in Complex-Activity-Defs$ is called a *complex* activity node.

With *Activity-Def* we denote the activity definition type, i.e., the type extensionally specified by *Activity-Defs*.

5.3.10 Workflow Cooperation

We now specify how workflows *cooperate*, in particular workflows that are executed at *different* sites. The central assumption of the AGENTWORK cooperation model is that a workflow modeler specifying workflows at one site usually will not have detailed knowledge about the control and data flow of workflows running at another site. Thus, cooperation can only be modeled in a message-oriented manner by specifying with which remote workflow system or – more abstractly – with which cooperation partner which type of information shall be exchanged when. Any stronger workflow coupling – such as connecting nodes of different workflows directly by control or data flow edges – is viewed as inappropriate as this would require knowledge about the control and data flow of the cooperation partner.

Principally, workflow communication could be modeled by external data flow edges where the used object extensions represent some object “pool” shared by different workflow systems. However, as the handling of inter-workflow implications of control flow failures is one the topics of this thesis, AGENTWORK models workflow communication more explicit by special communication nodes. This has the advantage that the adaptation components can easier detect whether an adaptation affects cooperating workflows or not. Note that the handling of inter-workflow implications of control flow failures does *not* assume that the workflow management system AGENTWORK is used by *both* cooperation partners. It is only assumed that both cooperation partners use the same work-

flow communication model as described in the following.

5.3.10.1 Communication Nodes

To model workflow communication, AGENTWORK provides two additional node types, namely COMM-IN and COMM-OUT nodes. A COMM-IN node is placed within a workflow to specify that some information is expected from some other remote workflow system. A COMM-OUT node specifies that some information has to be sent to some other remote workflow system. The message content and the sender respective receiver is specified by so called *inter-workflow communication definitions* that can be assigned to COMM-IN or COMM-OUT nodes.

5.3.10.2 Inter-Workflow Communication Definitions

An *inter-workflow communication definition* is a tuple

$$(ws; o_1, o_2, \dots, o_n; c) \quad \text{where}$$

- ws identifies the cooperating workflow system that shall receive objects or that is expected to send objects.
- o_1, o_2, \dots, o_n are named object patterns according to 5.2.1 specifying the objects that shall be sent or that are expected to be received. These objects usually contain or describe (parts of) a product or service. For example, they may describe an electronic document containing an expertise needed by a cooperation partner (such as a chemotherapy report). They may also be describe an electronic letter stating that a product has now been delivered to a parcel service and will arrive at the cooperation partner during the next few days.
- c identifies the *Case* object to which the objects o_1, o_2, \dots, o_n belong (e.g., the patient or customer). The receiving workflow system uses c to identify the workflows needing the received objects o_1, o_2, \dots, o_n .

In case that an inter-workflow communication definition $(ws; o_1, o_2, \dots, o_n; c)$ has been assigned to a COMM-OUT node, this means that the objects o_1, o_2, \dots, o_n and c have to be delivered to ws when the control flow reaches the COMM-OUT node. The system ws then uses c to identify the workflows needing o_1, o_2, \dots, o_n . In case that an inter-workflow communication definition $(ws; o_1, o_2, \dots, o_n; c)$ has been assigned to a COMM-IN node, this means that it has to be inspected whether objects o_1, o_2, \dots, o_n have been received from ws for case c .

To a communication node, an arbitrary number of inter-workflow communication definitions can be assigned (e.g., to one COMM-OUT node two inter-workflow objects with *different* receiving workflow systems can be assigned). Neither users, programs or equipment can be assigned to communication nodes. Furthermore, communication nodes have to fulfill the same control flow constraints as activity nodes.

If n is a COMM-OUT or COMM-IN node, $comm-objs_n$ denotes the o_i of all inter-workflow communication definitions assigned to n . The elements in $comm-objs_n$ are called *communication*

objects.

5.3.10.3 Communication Data Flow

The data flow w.r.t. communication nodes is defined similarly as for activity nodes. For a COMM-OUT node n , the communication objects $comm-objs_n$ are initialized by data flow edges having activity output objects or objects extensions as their source. Then, these communication objects are mapped via external data flow edges to object extensions (from where they are sent to remote workflow systems by some ecommunication infrastructure). Vice versa, for a COMM-IN node n , the elements in $comm-objs_n$ are initialized via external data flow edges from object extensions to which remote workflow systems have sent their data. Then, these communication objects are mapped to activity input or condition input objects or to object extensions. Thus, we only have to extend our definition of internal and external data flow in Section 5.3.6 by allowing elements of $comm-objs_n$ at any place where activity input objects, activity output objects, condition input objects, or object extensions are allowed.

Furthermore, data flow constraint 1 (input completeness) of Section 5.3.6.4 has to be extended in the sense that every communication object of a COMM-OUT node can be initialized properly by data flow edges. The data flow constraint 2 (forward-oriented data flow) has to be extended in the sense that also data flow edges having communication nodes as source or target have to be considered.

5.3.10.4 Workflow Definitions for Cooperation

A workflow definition W_{coop} for cooperation then is a triple

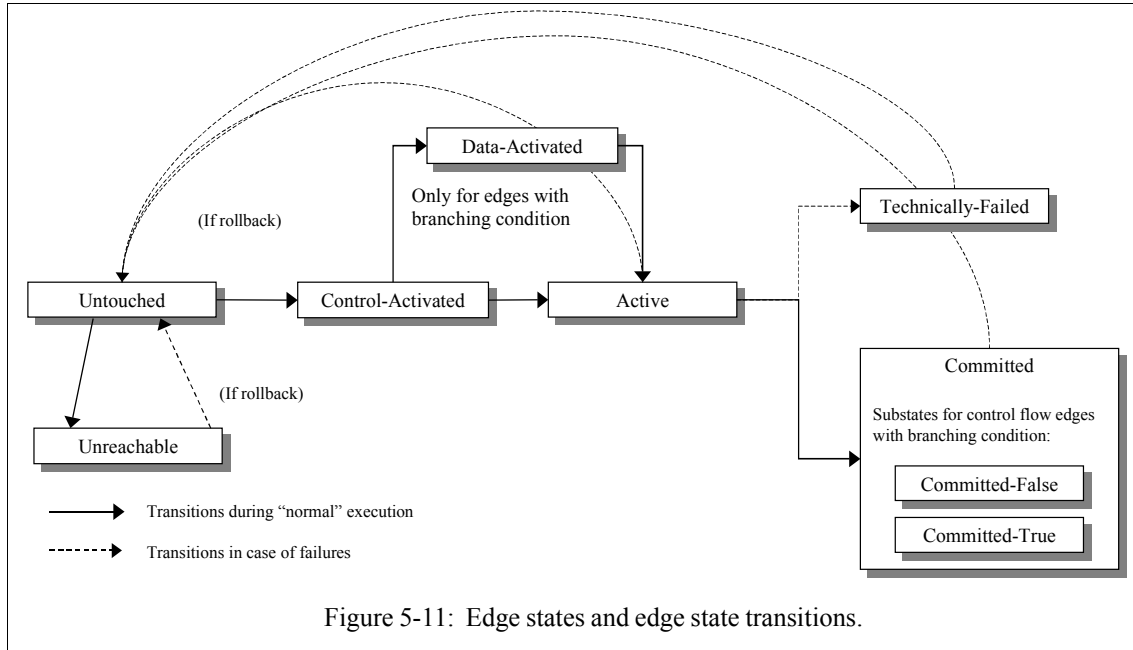
$$W_{coop} = (name, CF_{coop}, DF_{coop}) \quad \text{where}$$

- $name$ is an unique name for the workflow definition,
- WF_{coop} is a complex control flow as defined in 5.3.9 to which COMM-IN and COMM-OUT nodes with inter-workflow communication definitions have been added, and
- DF is a data flow definition according to 5.3.6 to which data flow edges having communication objects as their source or target have been added.

5.4 Workflow Execution Model

We now describe the AGENTWORK model of workflow *execution* by means of state transitions. A precise description of possible workflow execution states is necessary, as the handling of control flow failures affects *running* workflows and as the result of failure handling may depend on the particular states of nodes and edges at the moment of the failure. Furthermore, a state-based execution model allows to describe execution on a high level of abstraction, so that the failure handling process can be relieved from technical details.

We first introduce the different edge and node execution states (5.4.1-5.4.2). Second, we describe



how a workflow is executed by means of state transitions (5.4.3-5.4.6).

5.4.1 Edges States

A control or data flow edge e may be in the states *Untouched*, *Unreachable*, *Control-Activated*, *Data-Activated*, *Active*, *Committed*, *Committed-False*, *Committed-True*, or *Technically-Failed* (Figure 5-11):

- *Untouched* means that the control flow has not yet reached e .
- *Unreachable* means that the control flow will not reach e anymore (i.e., e will not be executed anymore, given that no rollback or loop back occurs). For example, for a control flow edge $e = (n, m)$ this is the case if n and m belong to a path of an OR-SPLIT/OR-JOIN block that does not qualify for execution.
- *Control-Activated* means that the control flow has reached e , i.e., the source node of e has committed.
- *Data-Activated* means that all input data needed for the execution of e are available. This state is only possible for edges with branching conditions (5.3.4), as this is the only edge type needing data for its execution. Note, that data flow edges provide data as a result, but do not need this data for their execution.
- *Active* means that the edge is currently executed.
- *Committed* means that the processing of e has been completed successfully, e.g., that the data

have been successfully mapped from the source node to the target node in case e is a data flow edge. For a control flow edge e with branching condition (i.e., $BC(e) \neq \text{NULL}$) we additionally introduce two sub-states to which e can commit: *Committed-True* means that $BC(e)$ has been evaluated to TRUE, while *Committed-False* means that $BC(e)$ has been evaluated to FALSE.

- *Technically-Failed* means that for some reason the processing of the edge failed. For example, an external data flow edge may be set to this state if the object extension from which data shall be retrieved is not accessible because of some database server crash. We call this state *Technically-Failed* to emphasize that it has nothing to do with *control flow* failures but corresponds to “technical” failures such as device, system, and transaction failures as described in 1.2.1. Nevertheless we have to consider such technical failures as they may imply that workflow duration estimations on which a workflow adaptation is based may not be met when the adapted workflow is continued.

The state of an edge e at a point in time t is returned by the function

$$\text{edge-state}(e, t). \quad (vi)$$

An edge e may be executed several times during the execution of a workflow instance I (due to loop iterations or rollbacks). Therefore, we define the function

$$\text{entry-of-edge-state}(I, e^i, s) \quad (vii)$$

which returns the point in time when an edge e enters a state s during its i -th execution w.r.t. a workflow instance I . If only the *current* or *next* execution of e is of interest, we may use the term $\text{entry-of-edge-state}(I, e^{\text{Current}}, s)$ or $\text{entry-of-edge-state}(I, e^{\text{Next}}, s)$ to get the point in time when e enters state s during this current respective next execution. If it is clear from the context which workflow instance or which execution of e is meant, we may simply omit the respective parameter, e.g., we may write $\text{entry-of-edge-state}(e^i, s)$ or $\text{entry-of-edge-state}(e, s)$.

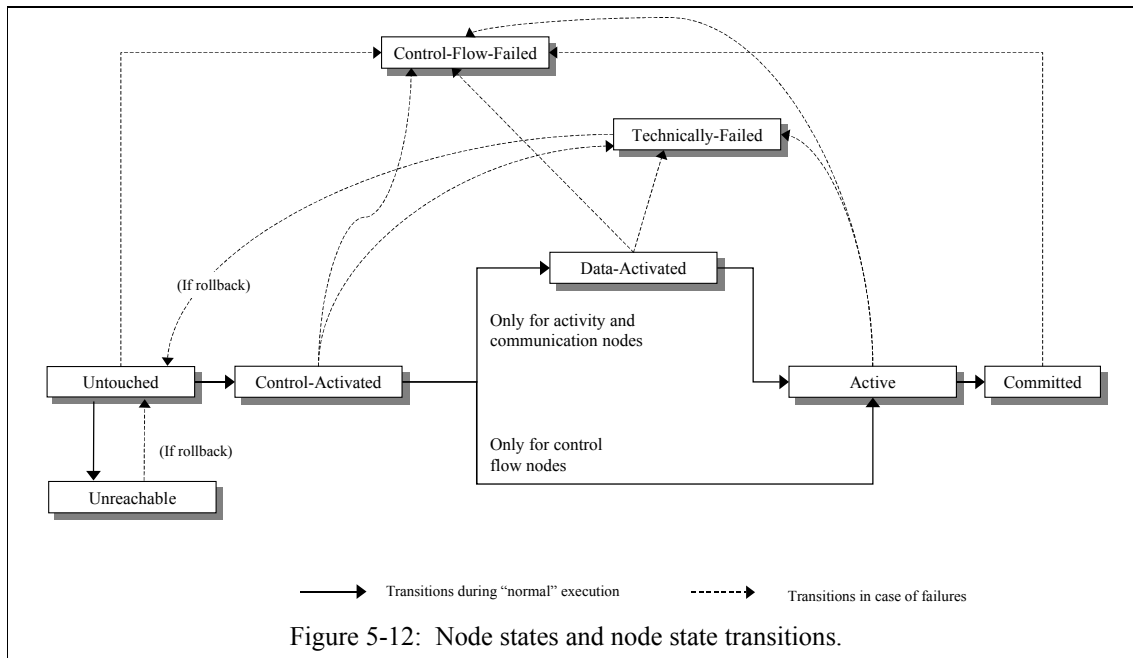
5.4.2 Node States

A control, activity or communication node n may be in the states *Untouched*, *Unreachable*, *Control-Activated*, *Data-Activated*, *Active*, *Committed*, *Technically-Failed* or *Control-Flow-Failed* (Figure 5-12):

- *Untouched* means that the control flow has not yet reached node n .
- *Unreachable* means that the control flow will not reach n anymore (given that no rollback or loop back occurs). For example, if a path after an OR-SPLIT node does not qualify for execution, all nodes of this path (up to the last node before the closing OR-JOIN node) are set to state *Unreachable*.
- *Control-Activated* means that the control flow has reached n , and – if n is a basic activity node – that the case and the resources needed to execute n have been assigned to n . A necessary condition for setting n to this state is that all incoming control flow edges have been set to

state *Committed*.

- *Data-Activated* means that all input data needed for the execution of n are available. This state is only possible for activity and communication nodes, as control nodes do not need data. Please recall that the condition evaluation at an OR-SPLIT node or a LOOP-END node is a matter of the associated control flow edges to which branching conditions have been assigned and which have the OR-SPLIT or LOOP-END node as their source node.
- *Active* means that the node is currently executed. For example, for a basic activity node n this means that the users or programs assigned to n are currently processing the activity definition assigned to n .
- *Committed* means that the processing of n has been successfully completed. In case of an activity or communication node this state also means that the output objects of n have been provided. A control node is set to state *Committed* directly after it has been set to state *Control-Activated* (as it does not process any data).
- *Technically-Failed* means that the execution of n is not possible because of technical reasons. A node n may be set to this state from the states *Control-Activated*, *Data-Activated*, or *Active*. For example, the transition *Control-Activated* \rightarrow *Technically-Failed* may occur when at least one of the incoming data flow edges of n is set to state *Technically-Failed* because of a database server crash so that n cannot be set to state *Data-Activated*.
- *Control-Flow-Failed* means that n is not adequate anymore because of a control flow failure.



As this thesis views control flow failures as an inadequacy of activities, this state is only possible for activity nodes. A node n may be set to this state from the states *Untouched*, *Control-Activated*, *Data-Activated*, *Active* and *Committed*.

A state transition *Committed* \rightarrow *Control-Flow-Failed* means that the activity $NAM(n)$ has retrospectively been identified as wrong from the control flow point of view. For example, after having administered a drug several times to a patient it may be detected that this patient has an allergy against this drug, so that the former activity nodes administering this drug must have to be viewed as *Control-Flow-Failed* retrospectively. The reason for setting these already committed activity nodes is to avoid that these nodes are executed again in case of a workflow rollback.

The question how a workflow with nodes in state *Control-Flow-Failed* can be set back to “normal” states is exactly the matter of control flow failure handling and the central topic of the following chapters.

The state of a node n at a point in time t is returned by the function

$$node\text{-}state(n, t). \quad (viii)$$

Analogously to edges, the point in time when a node n enters a state s during its i -th execution (during the execution of a workflow instance I) is returned by the function

$$entry\text{-}of\text{-}node\text{-}state(I, n^i, s). \quad (ix)$$

If only the *current* or *next* execution of n is of interest, we may use the terms $entry\text{-}of\text{-}node\text{-}state(I, n^{Current}, s)$ respective $entry\text{-}of\text{-}node\text{-}state(I, n^{Next}, s)$ to get the point in time when n enters state s during this current or next execution. If it is clear from the context which workflow instance or which execution of n is meant, we may simply omit the respective parameter, e.g., we may write $entry\text{-}of\text{-}edge\text{-}state(n^i, s)$ or $entry\text{-}of\text{-}edge\text{-}state(n, s)$.

5.4.2.1 Node State Synchronization Edges

Beside the control flow edges introduced in Section 5.3.5.1, it sometimes is useful to allow additional control flow dependencies between nodes. For example, let us assume that an oncological workflow consists of two parallel paths. One path administers cytostatic drugs while the other path performs monitoring activities for these cytostatic drugs. One of the monitoring activities must not be started before a particular drug administration in the other path has started, as the monitoring activity measures the side-effects of this drug administration. Formally, this means that a necessary condition for setting the monitoring activity node to state *Active* is that the drug administration node has been set to state *Active* as well. This cannot be achieved via the control flow edges of Section 5.3.5.1, as such a control flow edge $e = (n, m)$ has the semantics that m can only be set to state *Control-Activated* (and thus to *Active*) if n has been set to state *Committed*. Furthermore, it is forbidden by Criterion 2.

To support dependencies such as the one above without giving up the workflow structuring as

introduced in Criterion 5.3.5.3, we use so-called *synchronization edges* [REICHERT & DADAM 1998]: Given two nodes n_1 and n_2 and states s_1 and s_2 , a *synchronization edge*³ is a tuple

$$((n_1, s_1), (n_2, s_2)), \quad s_1, s_2 \neq \text{Untouched}, \text{Unreachable}$$

with the meaning that a necessary condition for setting n_2 to state s_2 is that

1. n_1 has been set to state s_1 , or that
2. n_1 has been set to state *Unreachable*.

By this, control flow dependencies such as the one in Figure 5-7 a) can be expressed without violating constraints 1-3 (Figure 5-13).

Condition 2 is necessary to avoid the “starvation” of n_2 if the path to which n_1 belongs will not be executed as it is, for example, a conditional path that does not qualifies for execution.

Additionally, it is not allowed that a synchronization edge has its source node within a LOOP-START/LOOP-END block, and its target node outside this LOOP-START/LOOP-END block, or vice versa. This is because it then would be unclear for which loop iteration the synchronization shall take place.

Formally, the control flow edges of Section 5.3.5.1 are special synchronization edges of type

$$((n_1, \text{Committed}), (n_2, \text{Control-Activated})) \quad (x)$$

For them, we still write (n_1, n_2) as a distinguishing notation as these edges express the “main-stream” of the control flow. In particular, in the following we term any edge of type (x) a *control flow edge* while any edge of a type different from (x) (such as $((n_1, \text{Active}), (n_2, \text{Active}))$) is termed *synchronization edge*.

To a synchronization edge $((n_1, s_1), (n_2, s_2))$ a *waiting condition* $WC = (min, max)$ may be assigned with the semantics that a necessary precondition to set n_2 to state s_2 is that since n_1 has been set to state s_2 at least the time specified by *min* and at most the time specified by *max* must have elapsed. Assigning a waiting condition $WC(e)$ to a synchronization edge e implicitly means that it has to be verified whether the *min* and *max* entries of $WC(e)$ are satisfiable at all (e.g., if e connects nodes of different paths of and AND-SPLIT/AND-JOIN block). In AGENTWORK, this sort of temporal verification is possible due to the temporal estimation algorithms introduced in Chapter 6 (*Workflow*

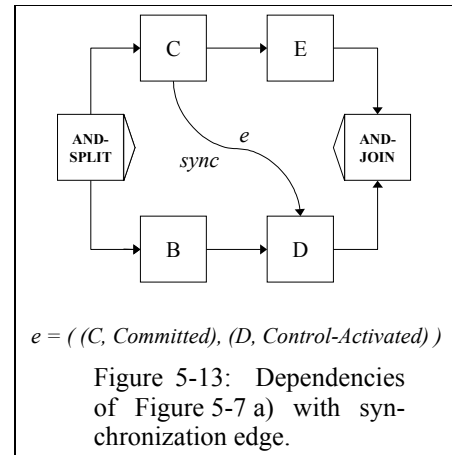


Figure 5-13: Dependencies of Figure 5-7 a) with synchronization edge.

3. Corresponding to a *soft* synchronization edge of [REICHERT & DADAM 1998].

Duration Estimation), but not further addressed as workflow verification is not our major topic.

It is not allowed to assign a *branching* condition to a synchronization edge as it otherwise would be unclear what to do if the branching condition is not TRUE. A synchronization edge may be in the states *Untouched*, *Unreachable*, *Active* and *Committed*.

5.4.3 Edge and Node Execution

We now describe how edges and nodes are executed. For this we first introduce some temporal tolerance parameters (5.4.3.1). Second, we describe the execution of the different edge types (5.4.3.2-5.4.3.4) and node types (5.4.3.5-5.4.3.6). Finally, we discuss transactional aspects of edge and node executions (5.4.3.7). As a preliminary remark we state that during workflow execution some edge or node types remain in some state only with a duration 0. This has only technical reasons to avoid too many case distinctions w.r.t. state transitions.

5.4.3.1 Tolerance Parameters

As workflows typically have a longer time span, the execution of nodes or edge should not be aborted immediately when technical difficulties occur. Rather, execution steps should be retried or difficulties should be repaired manually, if possible. To model this, AGENTWORK introduces two parameters *eng-time* and *user-time* of the duration type (4.3.2). They have the following semantics:

- *eng-time* specifies the maximal amount of time the workflow engine tries to perform a particular execution step (such as reading data from a source activity node or from a relational database). If the engine is not able to complete this execution step successfully during *eng-time* (as for example the source node has not been executed at all due to some conditional branching or as a database server has crashed), the engine requests a user to resolve the situation manually, e.g., by typing in the needed data.
- *user-time* specifies the maximal amount of time given to an “authorized” user to resolve the situation manually. The question which user is authorized is considered as a matter of implementation. For example, if the execution step deals with an activity node n , this user may be a member of $AUM(n)$ which may type in some missing data manually, or an administrator in case of some technical problem such as that an application program cannot be started. If the situation is not resolved manually during *user-time*, as for example the user is not available or is not able to resolve the situation manually, the execution step is assumed to have failed. The motivation for this parameter is that it is too restrictive to let an execution step fail without giving the workflow user or administrator the chance to intervene.

In particular, a node or edge is only set to state *Technically-Failed* after the time intervals specified by *eng-time* and *user-time* have elapsed without any successful attempt to complete an execution step. The particular values of *eng-time* and *user-time* and the question whether these values are global or may depend on the type of the execution step (e.g., edge or node execution) is left up to an implementation. Typically, *eng-time* should be not longer than several minutes, while *user-time* may last several hours, as it is the time scheduled for a manual process.

For more enhanced fault tolerance models concerning “normal“ workflow execution (i.e., without considering control flow failures), we refer to [WÄCHTER 1996, LIEBHART 1998].

5.4.3.2 Execution of Data Flow Edges

We describe the principal execution of data flow edges by the example of an *external* data flow edge. The other data flow edge types are executed analogously.

If $e = (s, t)$ is a reading external edge, e is set to state *Control-Activated* and then directly to *Active* when its target activity node or branching condition x (i.e., the node or condition x with $t \in \text{input}_x$) has been set to state *Control-Activated* (as in this state x needs its input data). During *eng-time* the engine tries to perform the object extension query specified by s and to map the retrieved data to t . If this data mapping fails as the database server has crashed or as the query returns an empty set because no object meeting the pattern expressed by t can be found, the engine requests an authorized to resolve the situation manually during *user-time*. For example, this user can try to restart the crashed database server, or can enter the requested data manually. If this also fails, e is set to state *Technically-Failed*. Otherwise, if the data can be mapped to t during *eng-time* or *user-time*, e is set to state *Committed*.

If $e = (s, t)$ is a writing external edge, e is set to state *Control-Activated* and then directly to *Active* when its source activity node n (i.e., the node n with $s \in \text{output}_n$) has been set to state *Committed* (as in this state n provides its output data). The further execution of e is analogously to the execution of the reading external edge.

All other data flow edge types are executed analogously so that we omit the description of their execution here.

Note that any edge $e = (s, t)$ is also set to state *Technically-Failed*, if the source object described by s does not match the pattern required by t according to 4.2.1.6, or if the currentness of the source object is not sufficient. For example, if h is an activity input pattern defined as

h : Hemato-Finding[parameter = Leukocyte-Count] NOT-OLDER-THAN (2, day)

(see 5.2.2), then $e = (s, h)$ would be set to *Technically-Failed* if s is a thrombocyte count or a leukocyte count older than 2 days.

5.4.3.3 Execution of Control Flow Edges

A control flow edge $e = (n, m)$ is set to state *Control-Activated* when n has been set to state *Committed*. In case *no* waiting or branching condition has been assigned to it, e is directly set to state *Active* and then to *Committed* (i.e., the duration of state *Active* is 0 in this case).

In case a **waiting condition** $WC(e)$ with *min* and *max* entries has been assigned to e , e is then directly set to state *Active* and remains in this state at least for the duration described by *min* and at most for the duration described by *max*. There are two principal possibilities to determine when the waiting time is over and e thus shall enter state *Committed*: First, the engine itself can select a point in time

$$t \in [\text{entry-of-edge-state}(I, e^{\text{Current}}, \text{Active}) + \text{min}, \text{entry-of-edge-state}(I, e^{\text{Current}}, \text{Active}) + \text{max}],$$

e.g., the half temporal distance (I being the workflow instance e belongs to). If an authorized user confirms t , the engine sets e to state *Committed*. Second, an authorized user himself selects a point in time t to indicate that the waiting time is over. In this case, the engine rejects this if $t < \text{entry-of-edge-state}(I, e^{\text{Current}}, \text{Active}) + \text{min}$, or generates an alert if the point in time $\text{entry-of-edge-state}(I, e^{\text{Current}}, \text{Active}) + \text{max}$ has been reached but the user has not selected such a t . It is considered as a matter of configuration which possibility is used. For the AGENTWORK workflow model it is sufficient to assume that the waiting constraint in some way is met.

In case a **branching condition** $BC(e)$ has been assigned to e , the engine executes every data flow edge of the set

$$S_e := \{(s, t) \in \text{Internal-Data-Flow} \cup \text{External-Data-Flow} \mid t \in \text{input}_e\} \quad (xi)$$

which is the set of all edges mapping data to an input object of e . If all edges in S_e have been set to state *Committed* (i.e., all objects in input_e could have been initialized), e is set to state *Data-Activated* (as all data needed for the condition evaluation are available). Then, e is set to state *Active*, and the condition $BC(e)$ assigned to e is evaluated. In case the condition is fulfilled, e is set to state *Committed-True*. In case the condition is *not* fulfilled, e is set to state *Committed-False*. In the latter case, all nodes and edges that cannot be reached anymore during the current workflow execution are set to state *Unreachable*.⁴ If at least one data flow edge in S_e has been set to state *Technically-Failed* or if the condition $BC(e)$ assigned to e could not have been evaluated, e is set to state *Technically-Failed*.

5.4.3.4 Execution of Synchronization Edges

A synchronization edge $e = ((n_1, s_1), (n_2, s_2))$ is set to state *Control-Activated* and then to *Active* if n_1 has been set to state s_1 . If *no* waiting condition has been assigned to e , it is then set directly to state *Committed*. If a waiting condition has been assigned to e , e is executed analogously to a normal control flow edge with waiting condition.

In the following we do not explicitly mention anymore that a necessary condition for setting any node n to any state s is: If a synchronization edge $e = ((m, s'), (n, s))$ exists, e must be in state *Committed* or in state *Unreachable*.

5.4.3.5 Execution of Basic Activity Node

We now describe how a basic activity node n with activity definition A is executed during the execution of a workflow instance I . As the execution of a basic activity node is a rather complex process, we order our description according to the different state transitions.

Transition *Untouched* \rightarrow *Control-Activated*: A node n is set to state *Control-Activated* when the

4. The node reachability algorithm is described in [BÖHME 2000].

control flow reaches n (i.e., when all incoming control flow and synchronization edges have been set to state *Committed* or *Committed-True*), and when the case for which n shall be executed and the required resources have been assigned to n . Concerning case and resource assignment, we are not interested in the details and in particular assume first that the case for which a node shall be executed is known when the control reaches n , and that second the needed resources are available. Therefore, we simply model this case/activity assignment via the functions

$$\begin{aligned} &case(I, n^i): Case, & \text{and} \\ &resources(I, n^i): Set<Resource>. \end{aligned} \tag{xii}$$

The function $case(I, n^i)$ returns the *Case* object that is assigned to the i -th execution of an activity node n during the execution of a workflow instance I . If it is clear from the context which workflow instance or which execution of n is meant, we may simply omit the respective parameter, e.g., we may write $case(n)$.

The function $resources(I, n^i)$ returns the *Resource* objects that are assigned to the i -th execution of an activity node n during the execution of the instance I . In contrast to the case assignment, the resource assignment is restricted by the function mappings AUM (*Activity* \rightarrow *User Mapping*), APM (*Activity* \rightarrow *Program Mapping*), and AEM (*Activity* \rightarrow *Equipment Mapping*), as defined in 5.3.3.1-5.3.3.3. For example, all *User* objects in the result set of $resources(I, n^i)$ have to match the patterns of the user definitions assigned to the activity definition of n by AUM . If it is clear from the context which workflow instance or which execution of n is meant, we may simply omit the respective parameter, e.g., we may write $resources(n)$.

Both functions will play an important role in Chapter 7 (*Control Actions*), as their return values determine whether a node execution is affected by a case-related respective a resource-related control action.

Transition *Control-Activated* \rightarrow *Data-Activated*: To set a node n from state *Control-Activated* to state *Data-Activated*, all data flow edges providing input data for n are executed. Formally, this is the set:

$$Input-Edges_n := \{(s, t) \in Internal-Data-Flow \cup External-Data-Flow \mid t \in input_n\} \tag{xiii}$$

If the execution of an edge $(s, t) \in Input-Edges_n$ fails, n is set to state *Technically-Failed*. The execution of (s, t) may fail as data could not have been read from an object extension, or as the source node that should have provided s has not been executed at all due to a conditional path. Another reason may be that s is not assigned to the same case which is assigned to n , i.e., if it holds $s.of \neq case(n)$. If all edges in $Input-Edges_n$ have committed, n is set to state *Data-Activated*, as all data needed to execute n are available now.

Transition *Data-Activated* \rightarrow *Active*: To set a node n from state *Data-Activated* to state *Active*, first all programs that have been assigned to n – via the function APM (*Activity* \rightarrow *Program Mapping*) as defined in 5.3.3.2 – have to be launched and provided with their input data. Second, to the users assigned to n , messages about the activity node to be executed are sent to the worklists of

these users. The node n is then set to state *Active*, when either a user assigned to n confirms that the activity execution has been started or when the first program assigned to n has been invoked successfully.

Transition *Active* → *Committed*: After the execution of the activity the output objects provided by the programs are mapped to the output objects of n . If at least one output object of n could not have been filled properly as for example one of the assigned programs has not been able to execute properly, n is set to state *Technically-Failed* as well. Otherwise, if all output objects of n have been filled properly, the execution of n is completed by setting it to state *Committed*.

5.4.3.6 Execution of Communication Nodes

A communication node is set to state *Control-Activated* when all incoming control flow and synchronization edges have been set to state *Committed* or *Committed-True*. The further processing is as follows:

For a COMM-OUT node n , all data flow edges initializing the communication objects of n are executed. If at least one of this edges is set to state *Technically-Failed*, n is set to *Technically-Failed*. If all edges initializing the communication objects have committed, n is set to *Data-Activated* and then directly to state *Active*. During the state *Active*, for each inter-workflow communication definition ($ws; o_1, o_2, \dots, o_m; c$) assigned to n , the communication objects o_1, o_2, \dots, o_m are delivered to the receiver system ws via the communication and integration layer. This is done *asynchronously*, as n is set to state *Committed* after the objects have been delivered and as the execution of the successor node of n is continued after this. If the successor node of n shall *not* be continued until ws sends a confirmation this has to be specified via a COMM-IN node placed directly after n .

In case that the delivery of the o_1, o_2, \dots, o_m is not possible during *eng-time* as for example ws is a workflow system not registered, the engine sends a message to an authorized user or the administrator to resolve the failure situation (e.g., to achieve that the ws entry is manually corrected). If the failure situation is not resolved during *user-time*, n is set to state *Technically-Failed*.

Vice versa, for a COMM-IN node n it is inspected for each inter-workflow communication definition ($ws; o_1, o_2, \dots, o_m; c$) assigned to n , whether the objects o_1, o_2, \dots, o_m have been sent for case c by ws and therefore are available. If at least one o_i is missing, the engine waits until a time described by *comm-in-time* has passed. The parameter *comm-in-time* specifies a duration typically significant longer than *eng-time* or *user-time*, as the time that should be waited until information from remote cooperation partners is received should be longer than the time it is waited until “internal” users respond. If there is at least one object missing after *comm-in-time* has expired, n is set to state *Technically-Failed*. If all objects are available, n passes the states *Data-Activated* and *Active* (both with duration 0) and is set to state *Committed*. The received objects are then mapped to other workflow objects or to object extensions via internal resp. external data flow edges.

5.4.3.7 ACID Transactions and Edge/Node Execution

In contrast to other approaches (e.g., [WÄCHTER & REUTER 1992]), AGENTWORK does not make

any assumption which steps during edge and node execution are enclosed within transactional borders, especially such following the ACID principle. This is left entirely up to an implementation. One could argue that it could make sense to enclose all steps executing a basic activity nodes (i.e., all steps between the states *Control-Activated* and *Committed*) within *one* ACID transaction. However, this does not make sense for many medical activities such as administering an 1 hour ETOPOSID infusion, as this activity cannot be rolled back from the moment at which it has been started (as the drug fluid cannot be extracted from the patient again). In particular, the demand to defer such “real actions” [GRAY & REUTER 1993] at the end of the transaction is not suitable for such medical activities as they mainly consist of such real actions, and not of database operations. Therefore, when it is said in the following that the execution of an edge or node has to be *aborted*, this does not necessarily mean a rollback of an ACID transaction, probably operated in the context of some 2-phase-commit protocol. It may also mean a *manual* abort, such as the immediate stop of an ETOPOSID infusion by an physician.

5.4.4 Block Execution

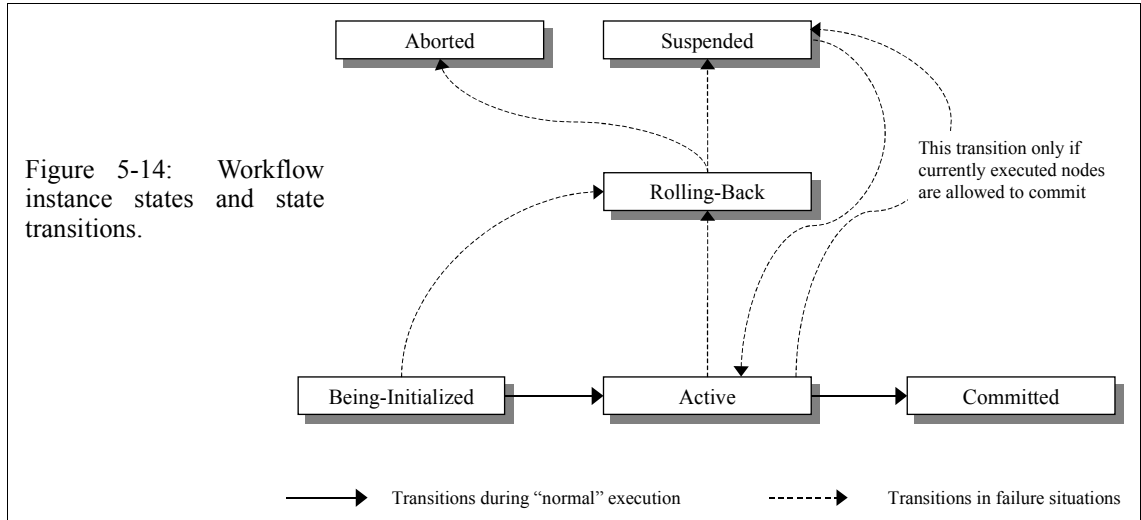
We now describe how the AGENTWORK engine executes control flow blocks. As an introductory remark we state that the opening node of a block (i.e., the AND-SPLIT, OR-SPLIT or LOOP-START node) is set to state *Control-Activated* when all incoming control flow and synchronization edges have been set to state *Committed* or *Committed-True*.

5.4.4.1 Execution of AND-SPLIT/AND-JOIN Blocks

The execution of an AND-SPLIT/AND-JOIN block with k parallel paths is performed as follows: After having been set to state *Control-Activated*, the AND-SPLIT node is directly set to state *Active* and then to state *Committed*. After this, the edges and nodes of the k parallel paths are executed. The closing AND-JOIN node is set to state *Control-Activated* when all edges between it and the last activity nodes of the k paths have been set to state *Committed*. Finally, the AND-JOIN node directly passes to the state *Committed*.

5.4.4.2 Execution of OR-SPLIT/OR-JOIN Blocks

The execution of an OR-SPLIT/OR-JOIN block with k conditional paths is performed as follows: After having been set to state *Control-Activated*, the OR-SPLIT node is directly set to state *Active* and then to state *Committed*. After this, the k outgoing control flow edges with branching conditions are evaluated. For each condition for which the condition evaluates to TRUE, the respective path is executed. For each condition for which the condition evaluates to FALSE, all path nodes and edges up to the closing OR-JOIN node are set to state *Unreachable*. The closing OR-JOIN node is set to state *Control-Activated* when all edges between it and the last activity nodes of the executed paths have been set to state *Committed*. Finally, the OR-JOIN node directly passes to the state *Committed*.



5.4.4.3 Execution of LOOP-START/LOOP-END Blocks

The execution of a LOOP-START/LOOP-END block is performed as follows: After having been set to state *Control-Activated*, the LOOP-START node is directly set to state *Active* and then to state *Committed*. After this, the path between the LOOP-START and the LOOP-END node is executed. Every time the path execution has been completed, the termination condition is evaluated. If the termination condition is FALSE, the loop path is executed again. If the termination condition is TRUE, the LOOP-END is set to state *Control-Activated*, and directly passes to state *Committed*. After this, the successor node of the LOOP-END node which is not identical with the corresponding LOOP-START node is executed.

5.4.5 Workflow Instance States

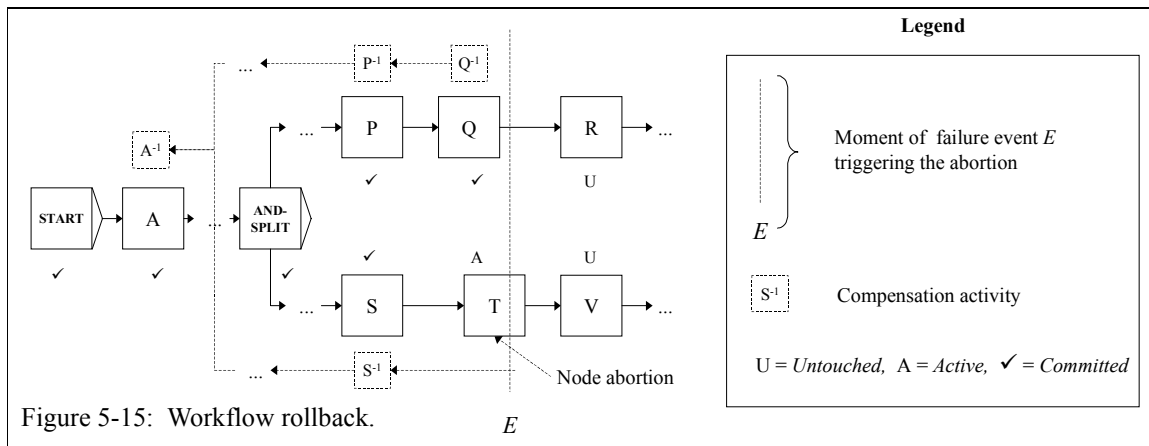
In the following we describe the states in which a workflow instance as a whole may be during its execution. These are the states *Being-Initialized*, *Active*, *Committed*, *Rolling-Back*, *Aborted* and *Suspended* (Figure 5-14). These states have the following meaning:

When a workflow definition shall be executed, an instance *I* based on this definition is generated and set to state *Being-Initialized*. In particular, the START node is set to state *Active*. During the state *Being-Initialized*, only “technical” procedures – such as setting all nodes (except the START node) and edges to state *Untouched* – are performed so that we omit details here. If at least one initialization procedure could not have been executed properly, the START node is set to state *Technically-Failed* (see below for the further handling of *I* in case that a node failed technically). If all initialization procedures have been executed successfully, the START node is set to *Committed*, and *I* is set to state *Active*. Then, the edges, nodes and blocks of the workflow are executed as described in 5.4.3 and 5.4.4. When the END-node has been reached, *I* is set to state *Committed*.

During the states *Being-Initialized* and *Active*, *I* may be set to one of the “failure” states *Rolling-Back*, *Aborted*, and *Suspended*. For example, the state transition sequence *Active* → *Rolling-Back* → *Aborted* may be performed for *I* if an *abort* control action has been triggered by some control flow failure (see Section 3.3.1), or if at least one node or edge of *I* has been set to state *Technically-Failed*.⁵ The state *Rolling-Back* means that *I* is rolled back to some already committed nodes. In particular, first the execution of edges and nodes in state *Active* (e.g., of the *T*-node in Figure 5-15) is aborted (also see 5.4.3.7). Second, for all nodes in state *Committed* compensating activities (5.3.2) are processed in the reverse order of the executed control flow (Figure 5-15).

Alternatively, instead of being set to state *Aborted*, *I* may be set to state *Suspended* if it makes sense to continue the workflow after a while. For example, a chemotherapy workflow may have to be suspended because the patient has got a hematological toxicity and has to recover for one week before the chemotherapy can be continued. Before setting *I* to state *Suspended*, a rollback to the last committed nodes may become necessary if nodes in state *Control-Activated*, *Data-Activated*, or *Active* are not allowed to commit (state transition sequence *Active* → *Rolling-Back* → *Suspended* in Figure 5-14). For example, this may be because nodes in state *Active* administer drugs that are responsible for the toxicity and thus have to be aborted immediately. When the workflow instance is continued after the suspension, it is set back to state *Active*.

The precise criteria stating which types of control flow failures lead to rollbacks, abortions or suspensions of workflows are given in Chapter 7 (*Control Actions*). Furthermore, as we focus on the dynamic adaptation of workflow due to local control flow failures, and not an workflow transaction aspects, we refer to the authors already mentioned in Chapter 2 (*Related Work*), in particular [GREFEN ET AL. 1999 B, KAMATH & RAMAMRITHAM 1998, DAVIS ET AL. 1996, ALONSO ET AL. 1996, LEYMAN 1995] for more technical details on workflow abortion and compensation.



5. Recall from 5.4.3 that a node is only set to state *Technically-Failed* if short-term attempts (e.g., during the next hours) to resolve the technical failure already have failed.

5.4.6 Execution of Complex Activity Nodes

A node n with complex activity definition C is executed as follows: When n has been set to state *Control-Activated*, a workflow instance I_C based on the workflow definition W_C assigned to C is generated and synchronously executed as described in 5.4.5. During the execution of I_C , input objects from n are mapped to activity nodes or control flow edges of I_C according to the data flow definition. Analogously, output object from activity nodes of I_C are mapped to the output objects of n . When I_C is set to state *Committed*, also n is set to state *Committed*, and the super-workflow to which n belongs continues with the successor node of n .

We emphasize that I_C is viewed as a *part* of its super-workflow with composite semantics. In particular, if the super-workflow is set to one of the failure states, this is also done for the sub-workflow. In contrary to this, this does not necessarily hold for the other direction. For example, if I_C is set to state *Suspended* because of some control flow failure, other paths of the super-workflow (i.e., those paths not containing the node with the complex activity definition C) may be continued.

5.5 Related Work

We now compare the workflow model of AGENTWORK with other workflow models in terms of goals 1-5 of 5.1. We concentrate on *petri nets* and *state/activity charts* as these are some of the most frequently language types used for workflow management. For other language types relevant for workflow management we refer to the literature (e.g., [BAETEN & WEIJLAND 1990] for process algebras and [SENKUL ET AL. 2002, DAVULCU ET AL. 1999] for transaction logics).

5.5.1 Petri Net-Based Workflow Modelling

Petri nets have been introduced as a calculus for the formal description of dynamic systems such as nuclear power plant control systems or computer networks [PETRI 1962]. Formally, a petri net is a bipartite graph consisting of a set T of so-called *transition* nodes, a set P of so-called *place* nodes and a set of *edges* $E \subseteq T \times P \cup P \times T$ connecting places and transitions. A transition usually represents a unit of work (i.e., an activity). A place typically represents an information “container” that can be marked with a set of so-called *tokens* representing data objects which are used, modified, or created by transitions. For example, in a medical domain a transition could model an examination or a drug administration, while tokens may represent patient data. A transition “fires” when for all input places the specified number of tokens is available. When the transition has been completed, its produced output tokens are moved to its output places. For further details concerning petri net syntax/semantics and the different petri net classes (e.g., place/transition, colored, hierarchical, stochastic and time-oriented nets), we refer to [BAUMGARTEN 1996, MURATA 1984].

Recently, petri nets also have been used for workflow modeling purposes in commercial workflow management systems (e.g., COSA [SOFTWARE LEY GMBH 2000], PROMATIS INCOME [OBERWEIS ET AL. 1994]) and workflow research projects (e.g., [QUAGLINI ET AL. 1999, ADAM ET AL. 1998, AALST 1998, OBERWEIS 1996]). This is because of the following reasons:

- First, their expressiveness (goal 1) is high. Especially colored and hierarchical nets provide a

broad range of constructs for control and data flow modeling suitable for workflow management. In particular, petri nets allow to integrate workflows with data distribution and communication processes. This means that not only the workflows themselves but also any communication of the workflows with users, application programs, databases, repositories and remote workflow systems can be modeled uniformly with transitions, places and tokens.

- Second, they provide a clear formal foundation (goal 3) supporting workflow *analysis*, such as reachability analysis or deadlock analysis [ADAM ET AL. 1998, AALST 1997].
- Third, several approaches have extended petri nets with temporal elements (*timed petri nets*) [ABDULLA & NYLÉN 2001, INABA ET AL. 1998, SCHÖF ET AL. 1995], e.g., to specify the minimal or maximal duration transitions. Thus, similar to waiting conditions introduced in 5.3.5.1, temporal constraints of workflow execution can be specified by using such timed petri nets.

However, there are several disadvantages so that petri nets have not been selected for this thesis:

- First, petri nets do not support object-oriented or object-relational data flow sufficiently (partial violation of goal 1). Though some work has been done to extend petri nets with object-oriented concepts (e.g., by modeling tokens as objects [LAKOS 1995]), results are of only preliminary nature yet. In particular, the question how to tailor analysis techniques to such object-oriented petri nets has not yet been answered clearly. Thus, as an object-oriented or object-relational data model is considered necessary for AGENTWORK, petri nets have not been selected.
- Second, goal 2 (adaptation-oriented workflow structuring) is not supported sufficiently. This is because the control flow and the data flow are not clearly separated as both are described by the same syntactical model elements (i.e., places, transitions and tokens). As in AGENTWORK adaptation primarily is an adaptation of *control flow*, this mixing up of control and data flow is not suitable for the purposes of this thesis.
- Third, goal 4 (adaptation-oriented execution support) is not met. In particular, the question which net parts have already been executed and which still have to be executed can only be answered by maintaining additional execution time information or by performing an extensive reachability analysis. This is a serious limitation, as the information about a workflows current execution state is essential for the dynamic adaptation process.

5.5.2 State/Activity-Charts

State/activity charts have been introduced by [HAREL 1987] as a formalism for specifying dynamic systems. So-called *activity charts* represent activities and their data dependencies while so-called *state charts* specify state transitions and thus the control flow between activities. For example, in Figure 5-16 the activity chart *MAIN_ACTIVITY* consists of the three activities A_0 , A_1 and A_2 , and specifies that A_1 and A_2 need the data elements $DATA_1$ respective $DATA_2$ from A_0 . The control flow between A_0 , A_1 and A_2 is specified by the state chart *CONTROL*. For example, *CONTROL* specifies that state Z_0 starts when activity A_0 has started ($/st!(A_0)$ for start activity A_0). Furthermore, it specifies that the state transition $Z_0 \rightarrow Z_0$ first has to be performed when event E_1 occurs with condition

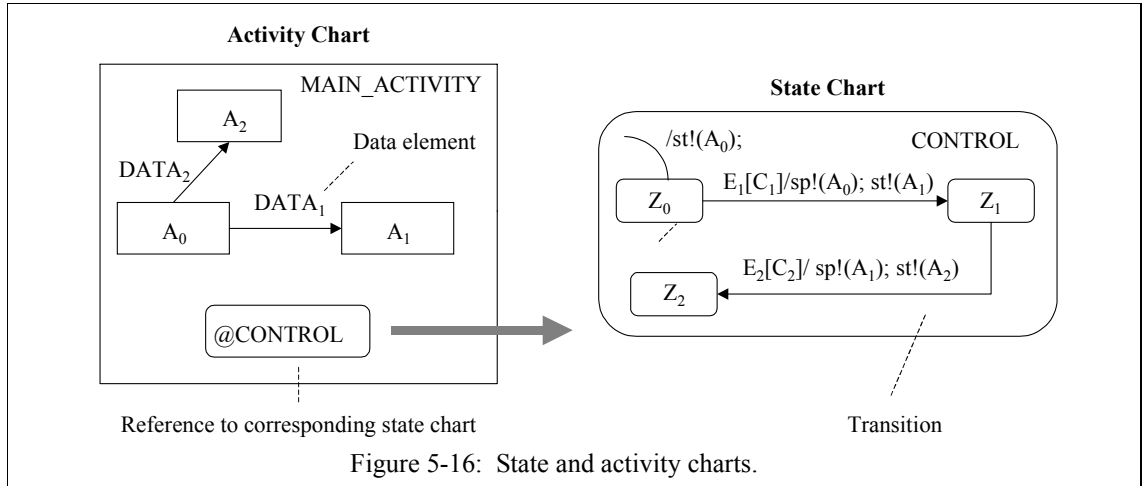


Figure 5-16: State and activity charts.

C_1 , and second consists of stopping A_0 ($/sp!(A_0)$ for stop activity A_0) and of starting A_1 ($/st!(A_1)$).

Because of their formal foundation and very strict distinction between control and data flow, state/activity charts have recently been used for process modeling in application development tools (e.g., STATEMATE/RHAPSODY [I-LOGIXS 2002], IBM OBJCHART) and in workflow research projects (e.g., [WODTKE & WEIKUM 1997]). However, their usage for AGENTWORK is limited as the control flow is “hidden“ in the state transition of the state charts. Thus, when a control action such as *drop(A)* would have been derived one would have to process all transition labels of the form $E[C]/st!(..)$ to identify where in the control flow A -nodes occur. Though this is possible, a block-oriented control flow definition as described in this chapter is much more suitable for the purposes of workflow adaptation as the control flow flow is represented much more explicitly.

5.6 Summary and Discussion

In this chapter we have introduced the AGENTWORK workflow model. Table 5-3 summarizes the characteristics of this workflow model and shows how they support the goals of 5.1. Particular strengths that go beyond the capabilities of other models such as the ADEPT_{FLEX} model [REICHERT 2000] include the fine-grained activity definitions and the full-fledged object-oriented data model.

Generally, one may ask why external data-related actions such as database accesses or user inputs are linked to control (i.e., conditional) and data flow edges, and not to special data-providing activity nodes (as [REICHERT 2000] suggests). This requires that numerous definitions (e.g., for parameter supply and time computations) have to be provided not only for nodes, but also for edges. The main reason for this has been – in order to support an adaptation-oriented workflow structuring (goal 2) – to separate application activities such as administering a drug from “operational“ aspects such as data retrieval, as activities are the main objectives of adaptation, and not operational elements. Thus, the suggested workflow structuring directly supports workflow adaptation.

Several limitations have to be mentioned: First, the control and data flow constraints listed in

Goal		Supported by
1	Expressiveness	<ul style="list-style-type: none"> • Fine-grained high-level activity definitions on the basis of object-oriented data model (5.3.1) • Control node types for parallel, conditional and iterative execution (5.3.5) • Object-oriented data flow (5.3.6) • Node types for inter-workflow communication (5.3.10) • Synchronization edges (5.4.2.1)
2	Adaptation-Oriented Workflow Structuring	<ul style="list-style-type: none"> • Separation of activity definitions, control flow, and data flow (5.3.5-5.3.6) • Connected, symmetrical blocks without activity split/join (control flow constraints 1-3, (5.3.5.3) \Rightarrow Support of temporal workflow estimation • Hierarchical workflow modeling by sub-workflows (5.3.9) \Rightarrow Reusability of temporal sub-workflow estimation
3	Formal Foundation	<ul style="list-style-type: none"> • Logic-based data modeling (Chapter 4) • Set-oriented control and data flow definition (5.3.5-5.3.6) • Control and data flow constraints (5.3.5.3 and 5.3.6.4)
4	Adaptation-Oriented Execution Support	<ul style="list-style-type: none"> • Explicit edge, node and instance states (5.4.1, 5.4.2 and 5.4.5) • Explicit state transitions
5	Readability	<ul style="list-style-type: none"> • Connected, symmetrical control flow blocks (5.3.5.3) • Hierarchical workflow modeling by sub-workflows (5.3.9)

Table 5-3: Design goals and support by AGENTWORK workflow model.

5.3.5.3 and 5.3.6.4 can only ensure some basic forms of correctness, and cannot avoid all possible types of incorrect workflows. For example, AGENTWORK does not provide a mechanism to verify whether the conditions at an OR-SPLIT node guarantee that for any data constellation at least one path will qualify for execution. However, as workflow definitions have a higher complexity than typical procedural languages such as C++ or Java because of parallel path execution, and because program verification already cannot be solved in general for these language types [APT & OLDEROG 1994], one cannot expect a list of constraints ensuring an overall correct workflow definition. For enhanced work on workflow correctness and verification, see [REICHERT 2000, AALST 1997].

Second, AGENTWORK does not provide a full-fledged workflow transaction approach. Rather, to concentrate on the handling of *control flow* failures and dynamic adaptation, AGENTWORK incorporates only basic transactional elements such as compensating activities and rollbacks to support workflow abortion. For more enhanced aspects, such as forward recovery and cascading compensation to cope with data dependencies, we refer to the literature already mentioned in Chapter 2 (*Related Work*), e.g., [GREFEN ET AL. 1999 B, REUTER & SCHWENKREIS 1995, LEYMANN 1995].

Finally, though readability is supported by connected, symmetrical control flow blocks and hierarchical workflows, it is unlikely that the introduced workflow model will directly be usable by non-computer experts such as physicians. Thus, enhanced user interfaces visualizing workflows on a more abstract level and libraries of application-specific pre-defined workflow patterns are needed. However, this topic is beyond our scope so that we refer the literature (e.g., [AALST ET AL. 2000]).

In this chapter we describe how AGENTWORK estimates workflow durations. As already sketched in Chapter 3, duration estimations are mainly used in the context of *predictive* adaptation. As we will see in Chapter 8 (*Structural Adaptation Operators*), also the control flow operators invoked in the context of *reactive* adaptation will use workflow estimations under certain conditions in order to optimize structural adaptations. Thus, before going into the details of structural workflow adaptation, this Chapter 6 explains the AGENTWORK workflow estimation approach in detail. In particular, after having formalized “time” in Chapter 4 and workflow execution in Chapter 5, we have the formal instruments needed for this.

This chapter is organized as follows: In Section 6.1, we describe the principles of workflow estimation in AGENTWORK, and introduce some useful definitions. In Section 6.2 and Section 6.3, we describe how the durations of edge resp. node executions are estimated. In Section 6.4 we describe how node sequences and entire workflow blocks such as OR-SPLIT/OR-JOIN blocks are estimated. In Section 6.5 we describe how arbitrary control flow paths are estimated, and how such path estimations are used for predictive adaptation. In Section 6.6, we compare the AGENTWORK workflow estimation approach with related work. The chapter concludes with a summary and discussion in Section 6.7.

6.1 Estimation Principles and Definitions

In this section we introduce estimation principles and definitions. For this we first introduce which estimation *strategies* can be used (6.1.1). Second, we describe how estimation values for the dura-

tions of edge or node executions can be obtained in principle (6.1.2). Third, we introduce useful conventions and definitions (6.1.3).

6.1.1 Estimation Strategies: Average Case, Worst Case, Best Case

If a workflow execution duration shall be estimated, one can principally select between three estimation strategies, namely worst case estimation, best case estimation, and average case estimation. These different strategies are characterized now. In the following, VT again denotes the valid time interval of a control action ca , and P_{VT} the workflow part that is assumed to be executed during VT .

6.1.1.1 Worst Case Estimation

Worst case estimation means that for every edge, node, and block execution that duration is taken that is assumed to be the longest (i.e., "worst") one being possible. For example, if it is assumed that a certain drug infusion D will never take longer than 4 hours, then for worst case estimation the duration (4, hour) would be taken as estimation value for every D -node. In the context of predictive adaptation, estimating P_{VT} by worst case estimation has the advantage that adaptations caused by ca and applied to P_{VT} "usually" do not have to be taken back. This is because typically the affected workflow part is executed *faster* as estimated (i.e., a temporal acceleration as described in 3.4.5 occurs), so that P_{VT} does not become smaller. Thus, it usually cannot occur that a subpart of P_{VT} that has been assumed to be executed during VT now suddenly is executed beyond VT , which would mean that adaptations may have to be taken back. We say "usually" as of course worst case estimation cannot exclude that the execution of an edge, node, or block takes even longer than the assumed maximal duration. For example, it cannot definitely be excluded that the execution of a node based on the drug administration D will take 5 hours instead of 4 hours.

However, worst case estimation has the disadvantage that in order to satisfy ca often *further* adaptations have to be made when the workflow is continued after the adaptation. This is because often workflow parts which have not been considered so far now will be executed during VT as well (i.e., P_{VT} becomes larger due to faster execution), and thus may have to be adapted as well to satisfy ca .

6.1.1.2 Best Case Estimation

In contrast to worst case estimation, best case estimation means that for every edge and node execution that duration is taken which is assumed to be the shortest (i.e., "best") one being possible. Concerning advantages and disadvantages of best case estimation in the context of predictive adaptation, the situation is diametrical to worst case estimation: The main advantage of estimating P_{VT} by best case estimation is that "usually" *further* adaptations regarding ca do *not* have to be made when the workflow is continued after the adaptation. This is because typically the affected workflow part is executed *slower* than estimated (i.e., a temporal delay as described in 3.4.5 occurs), so that P_{VT} cannot become larger. We again say "usually" as best case estimation cannot exclude that the execution of an edge, node, or block is even faster than the assumed minimal duration.

However, best case estimation has the disadvantage that very often adaptations performed to satisfy ca and applied to P_{VT} have to be taken back, as workflow parts that have been assumed to be exe-

cuted during VT are now not executed during VT (due to the slower execution).

6.1.1.3 Average Case Estimation

Average case estimation takes a middle course between worst case estimation and best case estimation as for every edge, node, and block execution that duration is taken that is assumed to be the duration *on the average*. For example, if it is assumed that a certain drug infusion D will never take longer than 4 hours, but will always last at least 1 hour, then the arithmetic mean value of 2.5 hours could be taken as such an average duration. Obviously, in the context of predictive adaptation average case estimation has both the advantages *and* disadvantages of worst case estimation *and* best case estimation. But these advantages and disadvantages keep themselves in balance, i.e., it can be assumed that the necessity to take back an adaptation arises as often as the necessity of performing additional adaptations to satisfy *ca*.

6.1.1.4 Estimation Strategy of AGENTWORK

As already mentioned in 3.4.2, AGENTWORK in its current version uses average case estimation as default estimation strategy. There is no hard reason for using just exactly this estimation strategy. Rather, it is assumed that in practice the situations where adaptations have to be taken back and the situations where additional adaptations become necessary to satisfy a control action should keep themselves in balance. As this assumption may not be right we nevertheless describe all three estimation strategies, or, more precisely, describe how estimation values for maximal, minimal, and average execution durations can be defined and acquired. An estimation algorithm then performs worst case estimation, best case estimation, or average case estimation if it uses estimation values for maximal, minimal, resp. average execution durations. The final decision which estimation strategy shall be used for a workflow application can only be made when the three different estimation strategies and the adaptations based on them have been evaluated in practice.

Another reason why we describe all three estimation strategies is that in the context of workflow *cooperation* (Chapter 10) worst case estimation and best case estimation play an important role, too. This is because during such a workflow cooperation deadlines can be assigned to workflows to specify when a cooperation partner expects some result or service. Thus, when a structural workflow adaptation affects a workflow that shall provide such a result or service, worst and best case estimations can be performed to give the collaboration partner some information what this adaptation means in the best and in the worst case for deadline satisfaction.

6.1.2 Acquisition of Duration Estimation Values

Independently from the selected estimation strategy, any workflow estimation requires that estimation values exist at least for the execution durations of edges and nodes. As in AGENTWORK workflow estimation plays a central role especially for predictive adaptation, these values have to be of a high quality, as a wrong estimation may require that adaptations have to be taken back or have to be performed additionally to satisfy a control action. Thus, AGENTWORK takes care very much of providing realistic estimation values. As roughly sketched in Chapter 3, AGENTWORK supports two

principal possibilities to obtain estimation values, namely duration estimations at definition time and duration measurements at execution time.

6.1.2.1 Duration Estimations at Definition Time

This means that the workflow modeler assigns estimation values to edges and nodes of a workflow definition. For some workflow applications, this is possible as often textual process descriptions specify how long some activities shall last. For example, most hematological therapy guidelines (e.g., [HAVEMANN 1994, PFREUNDSCUH & LÖFFLER 1994, DIEHL 1993, RIEHM 1995]) specify the duration of drug infusions precisely. This is because in order to achieve a maximal therapeutic effect and minimal side-effects, strict drug time tables have to be met.

In order to facilitate such duration estimations at definition time, it often makes sense to group edges and nodes in a workflow definition, and to specify estimation values only for these groups. For example, concerning nodes it makes sense to group nodes by their assigned activity definition, and to assign estimation values only to these groups where all node of a group have the same activity definition. The particular grouping criteria will be described in the sections on edge and node execution durations (6.2 and 6.3).

6.1.2.2 Duration Measurements at Execution Time

It is clear that it will not be possible for all workflow applications to specify representative and realistic estimation values at workflow definition time. This is because execution durations often will be influenced by events occurring during execution, such as a delayed database server response or the absence of a workflow user in the moment when an activity is presented in the worklist of this user. It is unlikely that the influence of such events on execution durations can be estimated at workflow definition time. One way to cope with this is to perform long-term measurements of the durations of edge and node executions for every executed workflow by the workflow engine. If these measurements have been performed and recorded for a longer time (e.g., for 6 months or a year), estimation values based on these measurements can be assumed to be much more representative than duration estimations obtained at workflow definition time. Thus, estimations about future workflow executions can be based on these measurements. Analogously to duration estimations at definition time, the measurements may be grouped, e.g., the execution durations for activity nodes may be grouped by the activity definitions. In Section 6.1.3.2, we will precisely describe how estimation values can be obtained from such (grouped) execution measurements.

The idea to measure and store execution durations has already been implemented in several workflow management systems. For example, in order to detect bottlenecks and thus to optimize workflows or workflow environments, the SAP WORKFLOW system provides the so-called SAP BUSINESS INFORMATION CENTER which stores and groups execution data such as activity execution durations in a multidimensional way [BERTHOLD ET AL. 1999]. By this multidimensional grouping, it can be derived for instance how long it takes a certain department or staff member to execute a certain activity type on the average. AGENTWORK adopts this idea of measuring execution durations and grouping them in a multidimensional manner, and extends this idea for the specific purposes of workflow estimation and adaptation. As already mentioned in Chapter 3, a useful

combination of these two principal possibilities of obtaining estimation values is to use duration estimations specified at workflow definition time for the first phase of an AGENTWORK installation, and then to continuously refine them by temporal measurements performed at execution time.

6.1.3 Conventions and Definitions

In this section we introduce some useful conventions and definitions, namely for the significance of durations (6.1.3.1), for estimation values (6.1.3.2), and for state transition durations (6.1.3.3).

6.1.3.1 Significance of Durations

In practical workflow management, the question which durations have to be considered depends on the average execution durations of all edges and nodes viewed together. For example, if for a workflow application the execution of an external data flow edge retrieving data from a database server takes only a few seconds on the average, this may be negligible if the average execution durations of activities in this workflow application are hours. However, this may not be negligible if the average execution durations of activities are minutes or even seconds (e.g., if all activities are processed entirely by programs without any manual interaction). Thus, when we say that a duration is *significantly* longer or shorter than another duration, this is always meant in this application-specific way. In particular, all temporal relationship notations used for durations and points in time in the following, such as

$d_1 < d_2, d_1 \leq d_2$ etc., $t \in [t_1, t_2], t_1 < t_2, t_1 \leq t_2$ etc. d_1, d_2 durations, t, t_1, t_2 points in time

implicitly consider this significance aspect, e.g., $t \in [t_1, t_2]$ means that point in time t is not significantly earlier than t_1 and not significantly later than t_2 .

To quantify significance, AGENTWORK allows to specify an application-specific threshold stating from when on a duration is significantly longer than zero.

6.1.3.2 Estimation Values

Let x be any workflow construct (e.g., an edge, node, or block) that is going to be executed in the future. Then, with

$dur(x)$

we denote the in-fact execution duration of x . The particular value of $dur(x)$ is unknown in advance. Thus, in correspondence to the three estimation strategies described in 6.1.1, we need estimation values to predict the execution duration of x . We define:

$dur-av(x)$: assumed average execution duration of x (for average case estimation), (i)

$dur-max(x)$: assumed maximal execution duration of x (for worst case estimation), (ii)

$dur-min(x)$: assumed minimal execution duration of x (for best case estimation). (iii)

If we use duration estimations at definition time as described in Section 6.1.2.1, these values have to be specified when the workflow is defined. If we use duration measurements at execution time as described in 6.1.2.2, these values can be derived as follows:

Having duration measurements d_1, \dots, d_n of n in-fact executions of x , we can set

$$dur-av(x) = \frac{1}{n} \sum_{i=1}^n d_i \quad (\text{arithmetic mean value of all } d_i) \quad (iv)$$

$$dur-max(x) = \max_{i=1 \dots n} \{d_i\} \quad (\text{absolute maximum of all } d_i) \quad (v)$$

$$dur-min(x) = \min_{i=1 \dots n} \{d_i\} \quad (\text{absolute minimum of all } d_i) \quad (vi)$$

Formulas (v) and (vi) have the disadvantage that outliers are weighted too much, as the largest resp. smallest d_i determines $dur-max(x)$ resp. $dur-min(x)$. To avoid this, AGENTWORK allows to use the “average” maximum and minimum, i.e.,

$$dur-max(x) = dur-av(x) + \sigma \quad (\text{“average” maximum of all } d_i) \quad (vii)$$

$$dur-min(x) = dur-av(x) - \sigma \quad (\text{“average” minimum of all } d_i) \quad (viii)$$

with $\sigma = \sqrt{\frac{1}{n} \sum_{i=1}^n (d_i - dur-av(x))^2}$ being the *standard deviation* of the d_i .

From the statistical point of view, (vii) and (viii) measure the average deviation of the d_i from their mean value “to the right” (vii) and “to the left” (viii), and thus reduce the influence of outliers.

As sketched in 6.1.2, it sometimes is more suitable to estimate duration values not for single edges or nodes but for a *group* of edges or nodes, if it can be assumed that the execution durations of the different group members do not differ significantly from each other. Then, we denote the average, maximal, and minimal duration of a group g as

$$dur-av(g) \quad \text{meaning: For all } x \in g \text{ it is assumed that } dur-av(x) = dur-av(g) \quad (ix)$$

$$dur-max(g) \quad \text{meaning: For all } x \in g \text{ it is assumed that } dur-max(x) = dur-max(g) \quad (x)$$

$$dur-min(g) \quad \text{meaning: For all } x \in g \text{ it is assumed that } dur-min(x) = dur-min(g) \quad (xi)$$

The question which grouping criteria should be used highly depends on the particular edge or node type, and therefore will be discussed in the resp. sections, i.e., 6.2 and 6.3.

6.1.3.3 State Transition Durations

Let x be a node and y a successor node of x (both in state *Untouched*), and let s, t be two node states. Then, we define

$$dur[x(s) \rightarrow y(t)] = \text{entry-of-node-state}(I, y^{Next}, t) - \text{entry-of-node-state}(I, x^{Next}, s), \quad (xii)$$

i.e., $dur[x(s) \rightarrow y(t)]$ denotes the duration between the next¹ point in time at which x is set to state s , and the next point in time at which y is set to state t (with I being the instance for which x and y are executed).

For arbitrary pairs x, y from the union of all nodes and edges (e.g., x being a node and y an edge), and node resp. edge states s, t , the term (xii) is defined analogously (i.e., by using the function *entry-of-edge-state* instead).

In the following sections 6.2–6.4, we will describe how the durations of edge, node, block, and path executions are estimated.

6.2 Edge Execution Durations

According to Chapter 5, an edge passes several states during its execution. Formally, we can define the duration of an edge e as

$$dur(e) = dur[e(\text{Control-Activated}) \rightarrow e(\text{Committed})]. \quad (xiii)$$

Now, we have to estimate the values $dur\text{-}av(e)$, $dur\text{-}max(e)$ and $dur\text{-}min(e)$ as defined in 6.1.3.2. We first consider data flow edges (6.2.1 and 6.2.2), and then control flow and synchronization edges (6.2.3–6.2.5).

6.2.1 Edges for Internal Data Flow

In this case we can always assume that $dur(e) = 0$, so that $dur\text{-}av(e)$, $dur\text{-}max(e)$ and $dur\text{-}min(e)$ can be set to zero as well. This is suitable, as an edge e of the internal data flow only maps already available data either to an activity node or an edge with a branching condition.

6.2.2 Edges for External Data Flow

External data flow edges represent the data flow from or to data sources of the workflow environment, such as the data flow from or to a relational database or an user interface. As these data sources may be located anywhere in a distributed and heterogeneous environment, we cannot assume that the processing of such a data flow edge always has duration zero. This holds especially for external data flow edges which request data from an user interface. This is because the user

1. The *Next* parameter in (xii) is necessary as nodes may be executed more than once during a workflow execution (e.g., if located within a loop).

from whom the data is requested may not be present at the moment of the request so that some significant amount of time may pass until the requested data is entered. Generally, we only have to consider *reading* external data flow edges (5.3.6.1), i.e., where e is a data flow edge *reading* data from an object extension (the latter representing some data source). An edge e *writing* to an object extension has not to be considered, as its duration does not "directly" influence workflow execution as the execution of the control flow usually can be continued without waiting for the commitment of e .²

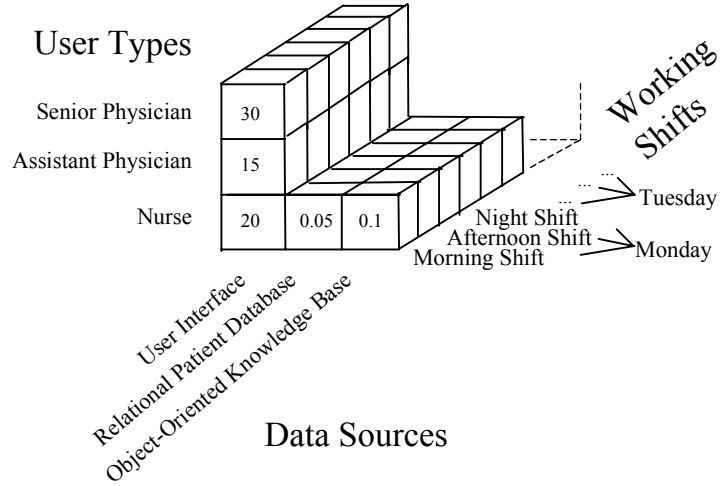
Independently from the way of obtaining estimation values $dur-av(e)$, $dur-max(e)$ and $dur-min(e)$ for *reading* external edges (e.g., by duration estimations at definition time or by duration measurements at execution time), AGENTWORK assumes that is not practicable and even not necessary to estimate $dur-av(e)$, $dur-max(e)$ and $dur-min(e)$ for *every* reading external data flow edge e in every workflow. Rather, AGENTWORK views it as sufficient to *group* all reading external data flow edges by the accessed data source type, and to estimate for each group g the group-specific values $dur-min(g)$, $dur-max(g)$ and $dur-av(g)$ which are then assumed to hold for every group element (as defined in 6.1.3.2). For example, such a group could consist of all reading external data flow edge accessing a certain relational database, and another group could consist of all reading external data flow edges accessing the user interfaces of a certain staff member class such as the physicians in a hospital. The rationale for such data source-oriented grouping is that typically the access durations will differ much more between *different* data sources than between different accesses to the *same* data source, and that furthermore the differences between the accesses of the same data source usually are negligible. For example, reading data from user interfaces usually will take significant longer than reading data from a relational database, and reading from a file-based system significant longer than reading from a relational database. In contrast to this, the durations of two reading accesses to the same relational database will not be significantly different.

If the estimation values are obtained by duration estimations at definition time, the workflow modeler consequently only has to specify the values $dur-av(g)$, $dur-max(g)$ and $dur-min(g)$ for each group g . If the estimation values are obtained by duration measurements at execution time, the measurements d_i in the formulas (iv)-(viii) consequently have to be the measurements of all so far executed group members.

Additionally, the edge executions may be grouped not only by the accessed data sources but also by day times and week days. For example, in many hospitals the morning shift between 8-12 a.m. usually is much more work- and data-intensive than other shifts in the afternoon or during the night. Thus, the execution of reading external data flow edges may take significantly longer during such a morning shift, as the data load of the workflow environment is higher. Furthermore, the execution duration may additionally depend on the week day. For example, many hospitals have fixed week days for surgical procedures or extended examinations, so that the data load may be higher during

-
2. Of course, a delayed or failed execution of a writing external data flow edge e may cause that a reading external data flow edge e' of a successor node that needs the data written by e cannot read this data, so that e' may be set to state *Technically-Failed* (5.4.3.2). However, such a failure cannot be considered by workflow estimation, but has to be handled by workflow monitoring.
-

Figure 6-1: Multidimensional grouping of execution durations (in minutes) for reading external data flow edges.



a Tuesday than during a Wednesday. In particular, data requests to physician interfaces make take significantly longer at such surgical or examination days than the physicians are much more absent from their desktops.

Thus, if such additional grouping criteria are used, this results in a multidimensional cube organization as shown in the example of Figure 6-1. The grouping dimensions *Data Sources*, *Working Shifts*, and *User Types* in this example are those used for the HEMATOWORK application. Each cube cell represents one group of reading external data flow edges (e.g., the group of all external data flow edge executions accessing assistant physician interfaces during the Tuesday morning shift) and stores the average execution duration (in minutes) of all group members.

One could argue that the cube of Figure 6-1 could be refined by additionally grouping the user interface accesses by each single physician or nurse. However, this is not appropriate as typically the particular user instance accessed by an external data flow edge e will often be unknown before e is in-fact executed. Thus, the estimation algorithm would not know which cell in the cube is the relevant one. Furthermore, such personalized duration measurements may affect data privacy and data protection [HERRMANN & BAYER 1998]. For example, in some countries it is not allowed to store how long a particular user needs for a particular task (such as entering data).

Obviously, the question which grouping dimensions shall be selected depends on the particular workflow application, and cannot be answered in general. Nevertheless, it is important that AGENT-WORK allows such a multidimensional grouping *at all* to obtain representative estimation values.

6.2.3 Unconditional Control Flow and Synchronization Edges

In this case we can always assume that $dur(e) = 0$, so that $dur-av(e)$, $dur-max(e)$ and $dur-min(e)$ can be set to zero as well. This is suitable as nothing really happens when such an edge is executed because no branching or waiting condition has been assigned to it.

6.2.4 Control Flow and Synchronization Edges with Waiting Condition

In this case a waiting condition has been assigned to e via the mapping $WC(e)$ introduced in 5.3. According to 5.3.4 (*Condition Definitions*), such a waiting condition consists of a *min* and a *max* entry specifying the minimal and maximal duration e may be in state *Active*.

If the estimation values are obtained by duration estimations at definition time, we set

$$dur-av(e) = \frac{min + max}{2}, \quad (xiv)$$

$$dur-max(e) = max, \quad \text{and} \quad (xv)$$

$$dur-min(e) = min. \quad (xvi)$$

Otherwise, if the estimation values are obtained by duration measurements at execution time, they have to be calculated according to the formulas (iv)-(viii). A grouping does not make sense for edges with waiting condition, as a waiting condition is very specific w.r.t. the source and target node connected by it, and typically not determined by other criteria.

6.2.5 Control Flow Edges with Branching Condition

Generally, the duration of an edge e with a branching condition (i.e., with $BC(e) \neq \text{NULL}$) cannot be assumed to be zero. This is because the input objects needed to evaluate the branching condition (see 5.3.4) may be provided by reading external data flow edges, that may not have duration zero (according to 6.2.2). Thus, if we assume that the condition evaluation itself has duration zero, the duration of an edge e with a branching condition is entirely determined by the edge set

$$S_e = \{x = (s, t) \in \text{External-Data-Flow} \mid t \in \text{input}_e\} \quad (xvii)$$

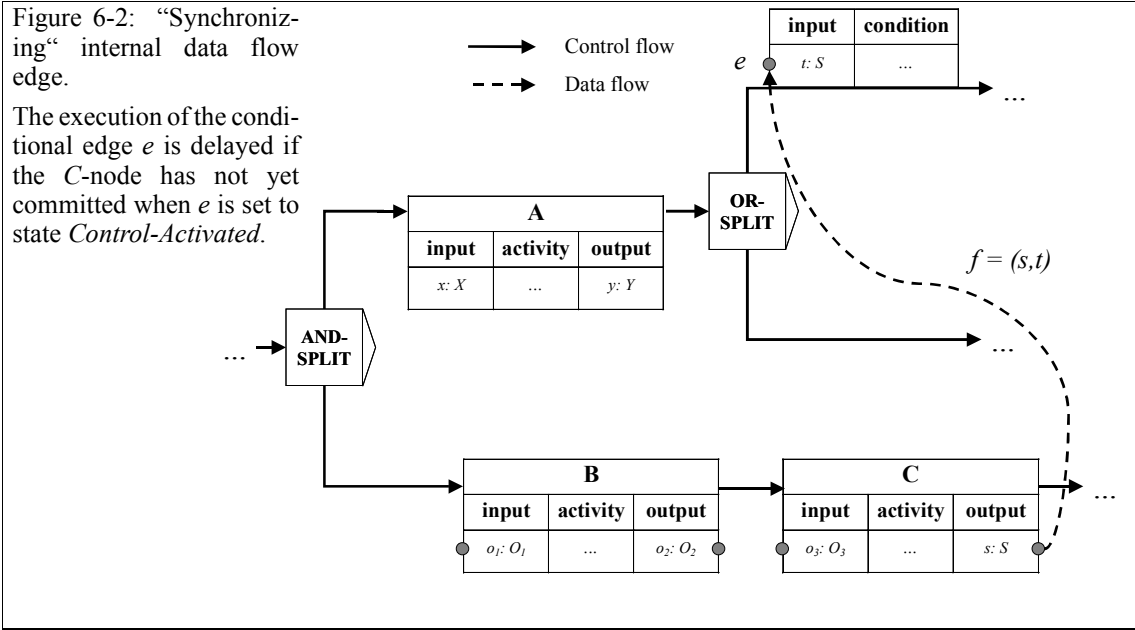
i.e., the set of all reading external edges mapping data from a data source to an input object of e . Then, independently from whether estimation values are obtained by duration estimations at definition time or by duration measurements at execution time, we set

$$dur-av(e) = \sum_{x \in S_e} dur-av(x) \quad (xviii)$$

(i.e., the average duration of e is the sum of the average execution durations of all edges in S_e).

$$dur-max(e) = \sum_{x \in S_e} dur-max(x) \quad (xix)$$

(i.e., the maximal duration of e is the sum of the maximal execution durations of all edges in S_e).



$$dur-min(e) = \sum_{x \in S_e} dur-min(x) \quad (xx)$$

(i.e., the minimal duration of e is the sum of the minimal execution durations of all edges in S_e).

Formulas (xviii)-(xx) hold for the pessimistic (and probably more realistic) assumption that in general the edges in S_e have to be processed *sequentially* (as, for example, S_e consists of external data flow edges that have to access the same database). If it can be assumed that the edges in S_e can be processed *in parallel*, one could use the maximum instead of the sum in (xviii)-(xx).

Note that formulas (xviii)-(xx) do *not* consider that an *internal* data flow edge f mapping data to one of the input objects of e may increase the duration e remains in state *Data-Activated* and thus may increase the execution duration of e significantly (as the source node of f has not yet committed when e is set to state *Control-Activated*; Figure 6-2). The necessary condition that such a situation may occur is that the source node of f is *not* a predecessor node of e . Such “synchronizing” internal data flow edges are not considered here because they are not a matter of e itself but of the workflow structure in the neighborhood of e . The handling of them will be discussed in 6.5 where the estimation of arbitrary control flow paths is described.

Edge Type		Duration Estimations	
		Duration Estimations at Definition Time	Duration Measurements at Execution Time
Edges for Internal Data Flow		Zero	Zero (measurements not done)
Edges for External Data Flow	Reading	Grouped by data source type, and optionally by other dimensions such as user types (in case of user interfaces), or day times and week days	
	Writing	Zero	Zero (measurements not done)
Unconditional Control Flow and Synchronization Edges		Zero	Zero (measurements not done)
Control Flow and Synchronization Edges with Waiting Condition		$dur-av(e) = \frac{min + max}{2}$ $dur-max(e) = max$ $dur-min(e) = min$ <p>Values of <i>min</i> and <i>max</i> according to condition definition in 5.3.4</p>	Calculated on the basis of execution duration measurements according to formulas (iv)–(viii)
Control Flow Edges with Branching Condition		<p>For $S_e = \{x = (s, t) \in External-Data-Flow \mid t \in input_e\}$:</p> $dur-av(e) = \max_{x \in S_e} \{dur-av(x)\}$ $dur-max(e) = \max_{x \in S_e} \{dur-max(x)\}$ $dur-min(e) = \max_{x \in S_e} \{dur-min(x)\}$	

Table 6-1: Estimation of edge execution durations.

6.2.6 Summary

As we will often refer to edge durations in the following, Table 6-1 summarizes how AGENTWORK estimates and measures the durations of control and data flow edges.

6.3 Node Execution Durations

According to Chapter 5, a node passes several states during a workflow execution. Formally, we can define the duration of a node *n* as

$$dur(n) = dur[n(Control-Activated) \rightarrow n(Committed)]. \quad (xxi)$$

Analogously to edges, we have to estimate the values $dur_{av}(n)$, $dur_{max}(n)$ and $dur_{min}(n)$ for the different node types. We first consider control nodes (6.3.1), then activity nodes (6.3.2), and finally communication nodes (6.3.3).

6.3.1 Control Nodes

For control nodes (i.e., nodes of type START/END, AND-SPLIT/AND-JOIN, OR-SPLIT/OR-JOIN, and LOOP-START/LOOP-END) we can always assume that $dur(n) = 0$, so that $dur_{av}(n)$, $dur_{max}(n)$ and $dur_{min}(n)$ can be set to zero as well. This is suitable as nothing time-consuming happens when such a node is executed. Please recall from 5.4.4 (*Block Execution*) that the condition evaluation at an OR-SPLIT or a LOOP-END node is *not* part of the execution of the OR-SPLIT or LOOP-END node itself. Rather, it is a matter of the associated control flow edges to which branching conditions have been assigned and which have the OR-SPLIT or LOOP-END node as their source node.

The execution duration of entire blocks enclosed by START/END, AND-SPLIT/AND-JOIN, OR-SPLIT/OR-JOIN, or LOOP-START/LOOP-END nodes, and in particular the execution delays caused by AND-JOIN or OR-JOIN nodes will be discussed in 6.4.

6.3.2 Activity Nodes

The execution of an activity node n can be divided into two phases which both have a duration significantly different from zero. The first phase starts at the moment when n is set to state *Control-Activated* and ends when n is set to state *Data-Activated* (data activation phase). This phase generally will have a duration significantly different from zero as reading external data flow edges (see 6.2.2) may have to be executed to provide all input objects for n . The second phase starts at the moment when n is set to state *Data-Activated* and ends when n is set to state *Committed* (working phase). This phase generally will have a duration significantly different from zero as it covers the work that has to be done according to the assigned activity definition. Thus, we can decompose the execution duration of n into

$$dur(n) = dur_1(n) + dur_2(n) \quad (xxii)$$

with

$$dur_1(n) = dur[n(\text{Control-Activated}) \rightarrow n(\text{Data-Activated})] \quad \text{and} \quad (xxiii)$$

$$dur_2(n) = dur[n(\text{Data-Activated}) \rightarrow n(\text{Committed})]. \quad (xxiv)$$

The question how estimation values $dur_{av_i}(n)$, $dur_{max_i}(n)$, and $dur_{min_i}(n)$ ($i = 1, 2$) can be obtained for $dur_1(n)$ and $dur_2(n)$ will now be discussed in the following two subsections.

6.3.2.1 Data Activation Phase

Concerning the data activation phase, $dur_{av_1}(n)$, $dur_{max_1}(n)$, and $dur_{min_1}(n)$ are obtained analo-

gously to control flow edges with branching conditions (see 6.2.5). This is because similar to control flow edges with branching conditions, the only time-consuming factor during the data activation phase of a node n are those reading external data flow edges initializing input objects of n . Thus, we can assume that $dur_1(n)$ is entirely determined by the edge set

$$S_n := \{x = (s, t) \in \text{External-Data-Flow} \mid t \in \text{input}_n\}. \quad (\text{xxv})$$

The values of $dur\text{-}av_1(n)$, $dur\text{-}max_1(n)$, and $dur\text{-}min_1(n)$ are then obtained analogously to control flow edges with branching conditions, i.e., by replacing S_e by S_n in the formulas (xviii)–(xx). Note that analogously to control flow edges with branching conditions, we do not yet consider that an *internal* data flow edge mapping data to one of the input objects of n may increase the duration n remains in state *Data-Activated* and thus may increase the execution duration of n significantly. This problem will be discussed in 6.5.

6.3.2.2 Working Phase

AGENTWORK assumes that for an activity node n the working phase duration of an activity node n does not depend on the particular workflow location of n but only on the activity definition A assigned to n via $NAM(n)$ ($NAM = \text{Node} \rightarrow \text{Activity Definition Mapping}$; see 5.3.9)³. Thus, we set

$$dur\text{-}av_2(n) = dur\text{-}av(A), \quad (\text{xxvi})$$

$$dur\text{-}max_2(n) = dur\text{-}max(A), \text{ and} \quad (\text{xxvii})$$

$$dur\text{-}min_2(n) = dur\text{-}min(A), \quad (\text{xxviii})$$

with $A = NAM(n)$ and $dur\text{-}av(A)$, $dur\text{-}max(A)$, $dur\text{-}min(A)$ being the average, maximal, and minimal duration that is assumed to be required to execute an activity specified by A (i.e., for all A -nodes the same values $dur\text{-}av(A)$, $dur\text{-}max(A)$ and $dur\text{-}min(A)$ are taken). We first assume that A is a *basic* activity definition according to 5.3.1. Second, we describe how to cope with A being a *complex* activity definition according to 5.3.8.

A = basic activity definition: If the estimation values are obtained by duration estimations at definition time, the workflow modeler consequently has to specify the values $dur\text{-}av(A)$, $dur\text{-}max(A)$, and $dur\text{-}min(A)$ for each basic activity definition. If the estimation values are obtained by duration measurements at execution time, the working phase durations of all A -node executions are measured for every basic activity definition A . Then, $dur\text{-}av(A)$, $dur\text{-}max(A)$, and $dur\text{-}min(A)$ are calculated according to the formulas (iv)–(viii) introduced in 6.1.3.2.

Analogously to the duration estimations for external reading data flow edges, further grouping criteria may be useful to obtain more precise estimation values $dur\text{-}av(A)$, $dur\text{-}max(A)$, and $dur\text{-}$

3. Note that this assumption does not hold for the data activation phase duration of an A -node. This is because the way the input data are provided may depend on the location of the node. For example, in one workflow an A -node may receive its input data from *external* data flow edges, in another workflow from *internal* data flow edges.

$\min(A)$. For example, for a medical application such as HEMATOWORK it makes sense to group the estimation values $\text{dur-av}(A)$, $\text{dur-max}(A)$, and $\text{dur-min}(A)$ by the user types and programs involved in the execution of an A -node. For instance, an experienced senior physician may be much faster with a bone marrow puncture than an assistant physician. Furthermore, for a chemotherapy administration a program calculating chemotherapy dosages typically will be much faster than a physician calculating the dosages manually.

At first glance, it does not make sense to additionally group the estimation values $\text{dur-av}(A)$, $\text{dur-max}(A)$, and $\text{dur-min}(A)$ by day times and week days, as it has been done for external reading data flow edges in 6.2.2. The argument for this is that for example in a medical application a bone marrow puncture will take the same amount of time both during the work-intensive morning shift and the afternoon shift, as the medical procedure that has to be performed does not depend on the day time or week day. However, this argument does not consider that the working phase as defined in (xxiv) does not only cover the time the execution remains in state *Active*. Rather, it also covers the time the execution remains in state *Data-Activated*. According to our activity execution model described in 5.4.3.5, a node execution enters state *Active*, when either a user assigned to the node confirms that the activity execution has been started or when the first program assigned to n has been invoked successfully. Thus, the duration a node may remain in state *Data-Activated* may be significantly longer during work-intensive working shifts. For example, during the morning shift a physician may have to perform 6-10 bone marrow punctures while only one or two punctures may have to be performed during an afternoon shift. Thus, not the puncture duration itself but the duration until the physician can start the next puncture (of those punctures in his worklist) may be significantly different during such different working shifts.

A non-medical example where activity durations may depend not on the day time but on the calendar month is the following: For software vendors there usually are one or two important exhibitions where they present new releases of their product. Due to such exhibition-related deadlines, workflow activities *not* related to the software development and preparation (such as writing internal reports or acquiring new projects) are often executed with a low priority during the months before the exhibition. Thus, due to the longer remaining of such activities in the state *Data-Activated* before such exhibition-related deadlines, the duration of the same activity may depend on the calendar month.

Therefore, if grouping criteria such as the one mentioned above are used, this results in a multidimensional cube organization of activity execution durations as shown in Figure 6-3. The dimensions *Basic Activity Definitions*, *Working Shifts*, and *User Types/Programs* of this example are those used for the HEMATOWORK application. Each cube cell stores the average execution duration (in minutes) of all node executions based on a particular basic activity definition and executed by a particular user type or program during a particular working shift. Again, we emphasize that the question which dimensions shall be selected depends on the particular workflow application, and cannot be answered in general. Nevertheless, it is important to note that AGENTWORK allows such a multidimensional grouping of estimation values *at all* to obtain representative estimation values.

A = complex activity definition: If A is a complex definition, this means that a workflow defini-

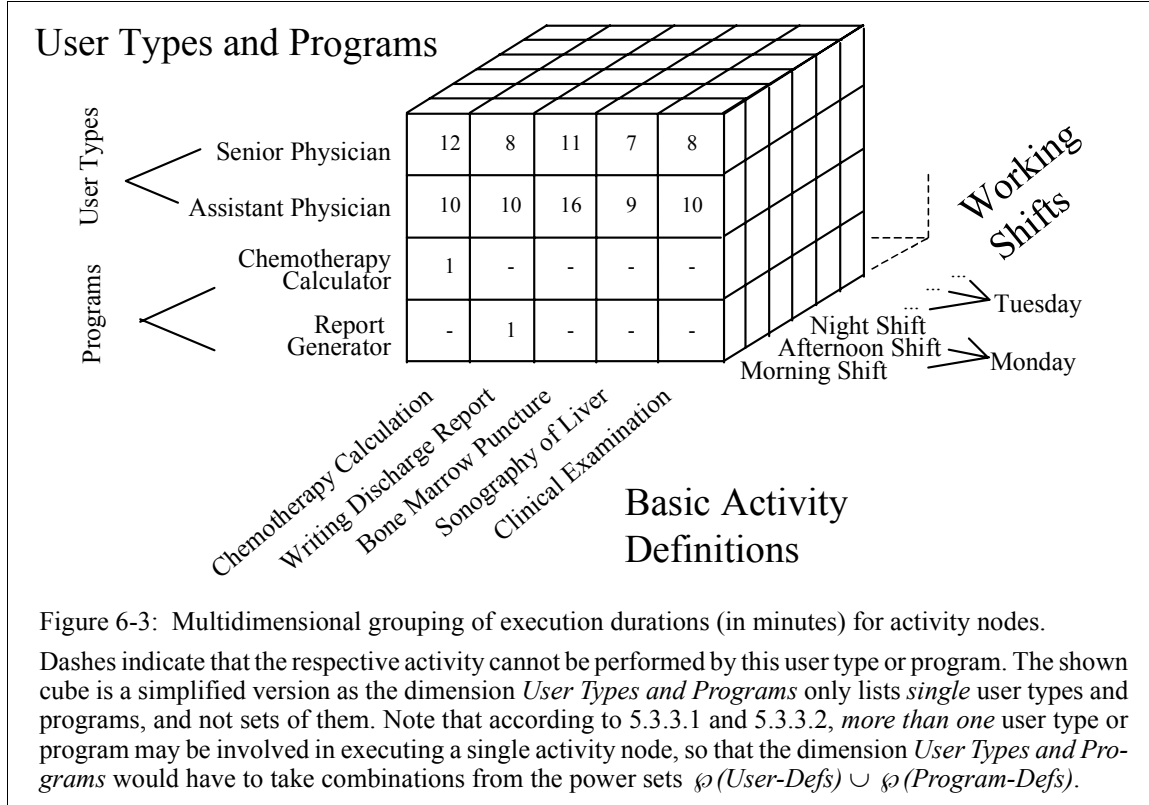


Figure 6-3: Multidimensional grouping of execution durations (in minutes) for activity nodes.

Dashes indicate that the respective activity cannot be performed by this user type or program. The shown cube is a simplified version as the dimension *User Types and Programs* only lists *single* user types and programs, and not sets of them. Note that according to 5.3.3.1 and 5.3.3.2, *more than one* user type or program may be involved in executing a single activity node, so that the dimension *User Types and Programs* would have to take combinations from the power sets $\wp(\text{User-Defs}) \cup \wp(\text{Program-Defs})$.

tion is assigned to A and executed when the control flow reaches an A -node. Thus, in this case the duration of the A -node has to be determined by estimating the duration of the whole workflow. However, this requires that workflow blocks and arbitrary control flow paths can be estimated which will be described in sections 6.4 and 6.5.

6.3.3 Communication Nodes

We recall from Chapter 5, that communication nodes in AGENTWORK specify when a workflow expects some information from some other workflow system (COMM-IN nodes), or when it sends information to some other workflow system (COMM-OUT nodes). Furthermore, we recall that the receiving or sending of the communication objects to COMM-IN or COMM-OUT nodes is done via data flow edges. Thus, similar to control flow edges with branching conditions or activity nodes, for a COMM-IN or COMM-OUT node n the duration of the phase from being set to state *Control-Activated* until entering the state *Data-Activated* cannot be assumed to be zero as external data flow edges may have to be processed to provide the communication objects.

In contrast to this, the duration a COMM-OUT node needs to deliver the received communication objects (i.e., $\text{dur}[n(\text{Data-Activated}) \rightarrow n(\text{Committed})]$) to remote workflow systems by inserting

them into some object extension for workflow communication, is assumed to have a negligible duration. This is because a COMM-OUT node is executed *asynchronously*, i.e., as soon as the communication objects have been delivered to object extensions for communication between workflow systems, n is set to state *Committed* and the execution of the sequence n belongs is continued. Thus, the time until the objects arrive at their destination does not delay the execution of the successors of n . Analogously, the duration a COMM-IN node needs to map the communication objects received from remote workflow systems to activity nodes, control flow edges with branching conditions or object extensions, is assumed to have a negligible duration as well.

Thus, the execution duration of COMM-IN or COMM-OUT node n is entirely determined by the set of the incoming reading external data flow edges retrieving communication objects, i.e.,

$$S_n = \{x = (s, t) \in \text{External-Data-Flow} \mid t \in \text{comm-objs}_n\}, \quad (\text{xxix})$$

with comm-objs_n denoting the communication objects assigned to the COMM-IN or COMM-OUT node n . The estimation values $\text{dur-av}(n)$, $\text{dur-max}(n)$, and $\text{dur-min}(n)$ then are obtained according to the formulas (xviii) - (xx) by replacing S_e through S_n .

Note that analogously to control flow edges with branching conditions and activity nodes, we do not yet consider that an *internal* data flow edge mapping data to one of the communication objects of n may increase the duration n remains in state *Data-Activated* and thus may increase the execution duration of n significantly. This problem will be discussed in 6.5.

6.3.4 Summary

Table 6-2 summarizes how AGENTWORK estimates and measures the durations of nodes.

6.4 Execution Duration of Sequences and Blocks

After having described how the execution durations of basic workflow elements such as edges and nodes can be obtained, we now can describe how more complex structures can be estimated. For this, we first describe how activity and communication sequences (6.4.1), AND-SPLIT/AND-JOIN blocks (6.4.2), OR-SPLIT/OR-JOIN blocks (6.4.3), and LOOP-START/LOOP-END blocks (6.4.4) can be estimated. In these sections, we assume that blocks are not nested, e.g., that a considered AND-SPLIT/AND-JOIN block does not contain an OR-SPLIT/OR-JOIN block. Furthermore, we assume that the nodes or edges in these sequences and blocks are *not* a target of synchronization edges or “synchronizing” internal data flow edges (as the one in Figure 6-2).

Second, we describe how the execution duration of arbitrarily nested blocks (6.4.5) and arbitrary control flow paths (6.5) is estimated.

6.4.1 Activity and Communication Sequences

Let $s = n_1 \rightarrow n_2 \rightarrow \dots \rightarrow n_k$ be a sequence of basic activity nodes or communication nodes. Let furthermore e_i be the control flow edge between n_i and n_{i+1} (as *no* n_i is an OR-SPLIT or LOOP-END

Node Type		Duration Estimations	
		Duration Estimations at Definition Time	Duration Measurements at Execution Time
Control Nodes		Zero	Not done (as duration assumed to be zero)
Activity Nodes	Data Activation Phase	For $S_n = \{x = (s, t) \in \text{External-Data-Flow} \mid t \in \text{input}_n\}$: $\text{dur-av}_I(n) = \max_{x \in S_n} \{\text{dur-av}(x)\}$ $\text{dur-max}_I(n) = \max_{x \in S_n} \{\text{dur-max}(x)\}$ $\text{dur-min}_I(n) = \max_{x \in S_n} \{\text{dur-min}(x)\}$	
	Working Phase	Grouped by activity definitions and optionally by other dimensions such as users types and programs, or day times and week days	
Communication Nodes		Analogously to data activation phase of activity nodes	

Table 6-2: Estimation of node execution durations.

node, e_i cannot have a branching condition). Then, we define the execution duration of s as

$$\text{dur}(s) = \text{dur}[n_I(\text{Control-Activated}) \rightarrow n_k(\text{Committed})]. \quad (\text{xxx})$$

This duration can be estimated as follows:

$$\text{dur-av}(s) = \sum_{i=1}^k \text{dur-av}(n_i) + \sum_{i=1}^{k-1} \text{dur-av}(e_i) \quad (\text{xxxi})$$

$$\text{dur-max}(s) = \sum_{i=1}^k \text{dur-max}(n_i) + \sum_{i=1}^{k-1} \text{dur-max}(e_i) \quad (\text{xxxii})$$

$$\text{dur-min}(s) = \sum_{i=1}^k \text{dur-min}(n_i) + \sum_{i=1}^{k-1} \text{dur-min}(e_i) \quad (\text{xxxiii})$$

6.4.2 AND-SPLIT/AND-JOIN Blocks

Let b be an AND-SPLIT/AND-JOIN block consisting of n parallel sequences s_1, s_2, \dots, s_n of basic activity nodes or communication nodes. Then, we can define the execution duration of b (AND-SPLIT $_b$ and AND-JOIN $_b$ denote the opening resp. closing node of b) as

$$\text{dur}(b) = \text{dur}[\text{AND-SPLIT}_b(\text{Control-Activated}) \rightarrow \text{AND-JOIN}_b(\text{Committed})] \quad (\text{xxxiv})$$

$$= \max_{1 \dots n} \{dur(s_i)\}$$

As the slowest sequence always determines the duration of the whole block, $dur(b)$ can be estimated as follows:

$$dur-av(b) = \max_{1 \dots n} \{dur-av(s_i)\} \quad (xxxv)$$

$$dur-max(b) = \max_{1 \dots n} \{dur-max(s_i)\} \quad (xxxvi)$$

$$dur-min(b) = \max_{1 \dots n} \{dur-min(s_i)\}. \quad (xxxvii)$$

6.4.3 OR-SPLIT/OR-JOIN Blocks

Let b be an OR-SPLIT/OR-JOIN block consisting of n conditional basic activity nodes or communication nodes. Let furthermore s_1, s_2, \dots, s_k ($k \leq n$) denote exactly those sequences that will be executed because the conditional control flow edges e_i ($i = 1 \dots k$) between the OR-SPLIT node and the first nodes of the sequences s_i ($i = 1 \dots k$) will commit to TRUE. Then, we can define the execution duration of b (OR-SPLIT _{b} and OR-JOIN _{b} denote the opening resp. closing node of b) as

$$\begin{aligned} dur(b) &= dur[OR-SPLIT_b(Control-Activated) \rightarrow OR-JOIN_b(Committed)] \quad (xxxviii) \\ &= \max_{1 \dots k} \{dur(s_i)\}. \end{aligned}$$

If those sequences s_1, s_2, \dots, s_k ($k \leq n$) that will be executed are known in advance, the duration $dur(b)$ can be estimated analogously as for AND-JOIN/AND-SPLIT blocks by using only the sequences s_1, s_2, \dots, s_k in the formulas (xxxv)-(xxxvii). Thus, the main problem is to determine those sequences s_1, s_2, \dots, s_k ($k \leq n$) that will be executed. We have to distinguish between two situations: First, that the estimation of the OR-JOIN/OR-SPLIT block occurs during “one-shot” predictive adaptation or iterative predictive adaptation with sub-intervals (6.4.3.1). Second, that it occurs during conditional iterative predictive adaptation (6.4.3.2).

6.4.3.1 Estimation of OR-JOIN/OR-SPLIT Blocks During One-Shot Predictive Adaptation or Iterative Predictive Adaptation with Sub-Intervals

We recall from Chapter 3, that for these two subtypes of predictive adaptation it is estimated which workflow part P_{VT} corresponds to a given valid time interval VT . During the estimation of P_{VT} , an OR-SPLIT node in state *Untouched* may be reached from which n conditional sequences s_1, s_2, \dots, s_n may start. As already sketched in 3.4.2, AGENTWORK then principally tries to predict which sequences s_1, s_2, \dots, s_k ($k \leq n$) starting at this untouched OR-SPLIT node will qualify for execution. This may be possible, if the data needed for determining which s_1, s_2, \dots, s_k ($k \leq n$) will qualify for execution are already available. Formally, this is done as follows: For every conditional edge e_i ($i = 1 \dots n$) between the untouched OR-SPLIT node and the first nodes of the sequences s_i ($i = 1 \dots n$) the two sets of all internal resp. external data flow edges retrieving data for e_i , i.e.,

$$I_{e_i} = \{x = (s, t) \in \text{Internal-Data-Flow} \mid t \in \text{input}_{e_i}\} \quad \text{and} \quad (\text{xxxix})$$

$$E_{e_i} = \{x = (s, t) \in \text{External-Data-Flow} \mid t \in \text{input}_{e_i}\} \quad (\text{xl})$$

are processed as follows: First, for every element $x = (s, t)$ of I_{e_i} it is checked whether

- a) the source node of s (i.e., the activity or communication node that provides s as output resp. communication object) is already in state *Committed* (as then s is available), and whether
- b) the currentness of s is sufficient w.r.t. the *NOT-OLDER-THAN* constraint of t (see 5.2.2).

Second, for every element $x = (s, t)$ of E_{e_i} it is checked whether

- c) the execution of the query described by s^4 returns an object of which the currentness is sufficient w.r.t. the *NOT-OLDER-THAN* constraint of t .

If for all e_i ($i = 1 \dots n$) and all elements of I_{e_i} and E_{e_i} these conditions a) and b) resp. c) hold, this means that all input objects needed to predictively evaluate the branching conditions for all e_i are available. By evaluating these conditions, it can then be determined predictively which sequences s_1, s_2, \dots, s_k ($k \leq n$) will qualify for execution. Thus, the duration of the OR-SPLIT/OR-JOIN block as defined in (xxviii) can be estimated analogously as for AND-JOIN/AND-SPLIT blocks by using only the sequences s_1, s_2, \dots, s_k in the formulas (xxv)–(xxvii).

If for at least one e_i ($i = 1 \dots n$) and at least one element in I_{e_i} or E_{e_i} conditions a) or b) resp. c) are not met, this means that at least one input object needed to predictively evaluate the branching condition assigned to e_i is not yet available. Thus, not all sequences s_1, s_2, \dots, s_k ($k \leq n$) that will qualify for execution can be determined but only a (possibly empty) subset s_1, s_2, \dots, s_l ($l < k$). The way AGENTWORK copes with this is to handle the other sequences s_{l+1}, s_2, \dots, s_n by reactive adaptation, as it cannot be estimated whether they will be executed during *VT* or not. Furthermore, if the question whether a successor node n of the OR-JOIN node will be executed during *VT* or not also depends on which of the sequences s_{l+1}, s_2, \dots, s_n will be executed in-fact, this node n also has to be handled reactively.

However, one problem remains: If condition c) holds for an e_i and an element $x \in E_{e_i}$, this does not exclude that an object retrieved by x when e_i is executed leads to the opposite condition evaluation result than the one derived at estimation time. For example, let us assume that the branching condition of an e_i states that the leukocyte count of the patient has to be not less than 2500, and that this leukocyte count must not be older than 24 hours. Let us furthermore assume that the OR-SPLIT/OR-JOIN node to which e_i belongs is estimated 18 hours before its actual execution, and that during this estimation the patient database query retrieving the leukocyte count provides a leukocyte count of 3000, so that the branching condition of e_i is predicted as true. However, when e_i is actually executed 18 hours later, the patient query may return a more recent leukocyte value of 2400

4. Recall from 5.3.6.1 that for reading external data flow edges (s, t) , s is a query on an object extension.

with the consequence that e_i commits to false. Such constellations cannot be avoided and are detected by workflow monitoring (see Chapter 9) which is able to detect that branching conditions have been predicted wrong.

6.4.3.2 Estimation of OR-JOIN/OR-SPLIT Blocks During Conditional Iterative Predictive Adaptation

We recall from Chapter 3, that for this subtype of predictive adaptation only the workflow part until the next *conditional* control node is estimated, i.e., until the next OR-SPLIT or LOOP-END node. For this workflow part, the workflow is adapted and continued. When workflow execution reaches the next OR-SPLIT or LOOP-END node, the remaining workflow part is estimated until the next conditional control node, and so on. By definition, this strategy has the consequence that for an OR-SPLIT node the sequences s_1, s_2, \dots, s_k ($k \leq n$) that will be executed are always known in advance so that no condition resolution has to be performed.

6.4.4 LOOP-START/LOOP-END Blocks

At first glance, the estimation of LOOP-START/LOOP-END blocks can be handled similar to OR-SPLIT/OR-JOIN blocks, as for LOOP-START/LOOP-END blocks conditions determine the execution as well. We recall from 5.3.5.1, that for a loop such a condition is assigned to the edge between the LOOP-END node and its successor node not being the LOOP-START node, and that this condition plays the role of the loop's termination condition. However, in contrast to OR-SPLIT/OR-JOIN blocks, LOOP-START/LOOP-END blocks form some sort of *iterative* conditional execution. In particular, the data needed to evaluate a loop's termination condition often will be provided by activity nodes *within* the loop sequence. This has the consequence that during a *single* loop iteration it may be possible to predict whether there may be another loop iteration or not (by using the same prediction mechanisms as used for condition evaluation in the context of OR-SPLIT/OR-JOIN blocks). However, these prediction mechanisms typically will be of no use to predict how *many* loop iterations will occur at all. For this, additional estimation values for the *number* of loop iterations become necessary. Thus, we first describe how to obtain such estimation values for the number of loop iterations (Section 6.4.4.1). Second, we describe how loop durations can be finally estimated on the basis of these estimation values for the number of loop iterations (Section 6.4.4.2). Note that these considerations are only relevant for estimations performed *not* under conditional iterative predictive adaptation. If conditional iterative predictive adaptation is the strategy during which the estimation is performed, this means that only the loop sequence is estimated until the LOOP-END node, then waited whether the loop will be executed once again, so that no prediction about the number of loop iterations is required for this strategy.

6.4.4.1 Estimation Values for Number of Loop Iterations

According to the two principal possibilities to obtain estimation information (duration estimations at definition time and duration measurements at execution time), two principal ways exist to obtain estimation values $it-num-av(l)$, $it-num-max(l)$, and $it-num-min(l)$ for the average, maximal resp. minimal number of iterations for a LOOP-START/LOOP-END block l .

First, at workflow definition time the workflow modeler specifies $it-num-av(l)$, $it-num-max(l)$, and $it-num-min(l)$ for every loop l in every workflow definition. For example, in hematooncology the *average* number of radiotherapy units that usually is necessary to achieve a permanent liver metastasis remission is known for many tumor types. Furthermore, a *minimal* number of radiotherapy units can be estimated on the basis of the minimal dosage that has to be applied to achieve a therapeutical effect at all. Analogously, a *maximal* number of radiotherapy units can be estimated on the basis of the dosage range that may cause toxic side effects and thus should not be entered.

Second, at execution time the number of iterations is recorded for every execution of every loop l . Then, the values $it-num-av(l)$, $it-num-max(l)$, and $it-num-min(l)$ are determined analogously to the formulas (iv)–(viii), i.e., by replacing $dur-av(x)$ by $it-num-av(l)$, $dur-max(x)$ by $it-num-max(l)$, $dur-min(x)$ by $it-num-min(l)$, and by using the recorded number of loop iterations as the d_i in the formulas (iv)–(viii).

For both ways, AGENTWORK assumes that a grouping of the values $it-num-av(l)$, $it-num-max(l)$, and $it-num-min(l)$ for instance by day times is not necessary. This is because AGENTWORK assumes first that the *number* of loop iterations does not depend for instance on day times, and second that any dependency of a loop's execution duration on day times or performing users or programs is entirely determined by the durations of the activities and edges in the loop sequence. For these activity and edge durations, grouping criteria have already been given in the sections 6.2 and 6.3.

6.4.4.2 Estimation of LOOP-START/LOOP-END Block

If values $it-num-av(l)$, $it-num-max(l)$, and $it-num-min(l)$ have been obtained for a loop l , the execution of the entire loop l can be estimated as follows: Let s denote the sequence of basic activity nodes or communication nodes between the LOOP-START and the LOOP-END node. Then, the duration of l can be estimated as follows:

$$dur-av(l) = it-num-av(l) \cdot dur-av(s) \quad (xli)$$

$$dur-max(l) = it-num-max(l) \cdot dur-max(s) \quad (xlii)$$

$$dur-min(l) = it-num-min(l) \cdot dur-min(s) \quad (xliii)$$

6.4.5 Nested Blocks

So far, we have assumed that blocks are not nested, e.g., that an AND-SPLIT/AND-JOIN block does not contain an OR-SPLIT/OR-JOIN block. We now describe to cope with a block b which contains at least one further block. The duration of such a nested block can be estimated in a straightforward manner by combining the estimation mechanisms described in sections 6.4.1–6.4.4 recursively. For example, let us assume that an OR-SPLIT/OR-JOIN block b is estimated, and that one of its sequences s that is assumed to be executed contains an AND-SPLIT/AND-JOIN block b' , i.e., that

$$s = n_1 \rightarrow \dots \rightarrow n_{j-1} \rightarrow b' \rightarrow n_{j+1} \dots \rightarrow n_k \quad (n_i \text{ being a basic activity or communication node}).$$

Then $dur(s)$ has to be estimated by replacing the resp. estimation value of n_j in the formulas (xxxi)–(xxxiii) by the resp. estimation value for block b' . As this has to be done analogously for all other nesting constellations, we omit a formal notation of estimation values for nested blocks.

6.5 Execution Duration of Arbitrary Control Flow Paths

We now describe how AGENTWORK estimates the execution duration of arbitrary control flow paths, or in other words, of a workflow part that does *not* form a complete workflow block. This is necessary, as for a control flow failure the failure node set (Section 7.4.1) describing the beginning of the workflow part to be estimated may consist of arbitrary nodes of the control flow, e.g., of nodes 3, 9, and 19 in Figure 6-4. As during predictive adaptation all paths starting from such a failure node set have to be estimated to see which nodes will be executed during the valid time interval of a triggered control action, we have to cope with such arbitrary control flow paths. We recall from Section 5.3.5.2 that a *control flow path* formally is any sequence $n_1 \rightarrow n_2 \rightarrow \dots \rightarrow n_k$ of activity, control or communication nodes (with using the symbol “ \rightarrow ” as an abbreviation for the control flow edge between two nodes). For example, in Figure 6-4 the two sequences

$$9 \rightarrow 10 \rightarrow 11 \rightarrow 12 \quad \text{and} \quad 9 \rightarrow 10 \rightarrow 13 \rightarrow 14 \rightarrow 15 \rightarrow 16$$

are control flow paths. Furthermore, for estimation purposes we also call a sequence which starts or ends with an edge, such as

$$\rightarrow 10 \rightarrow 11 \rightarrow 12 \quad \text{or} \quad 9 \rightarrow 10 \rightarrow 13 \rightarrow 14 \rightarrow$$

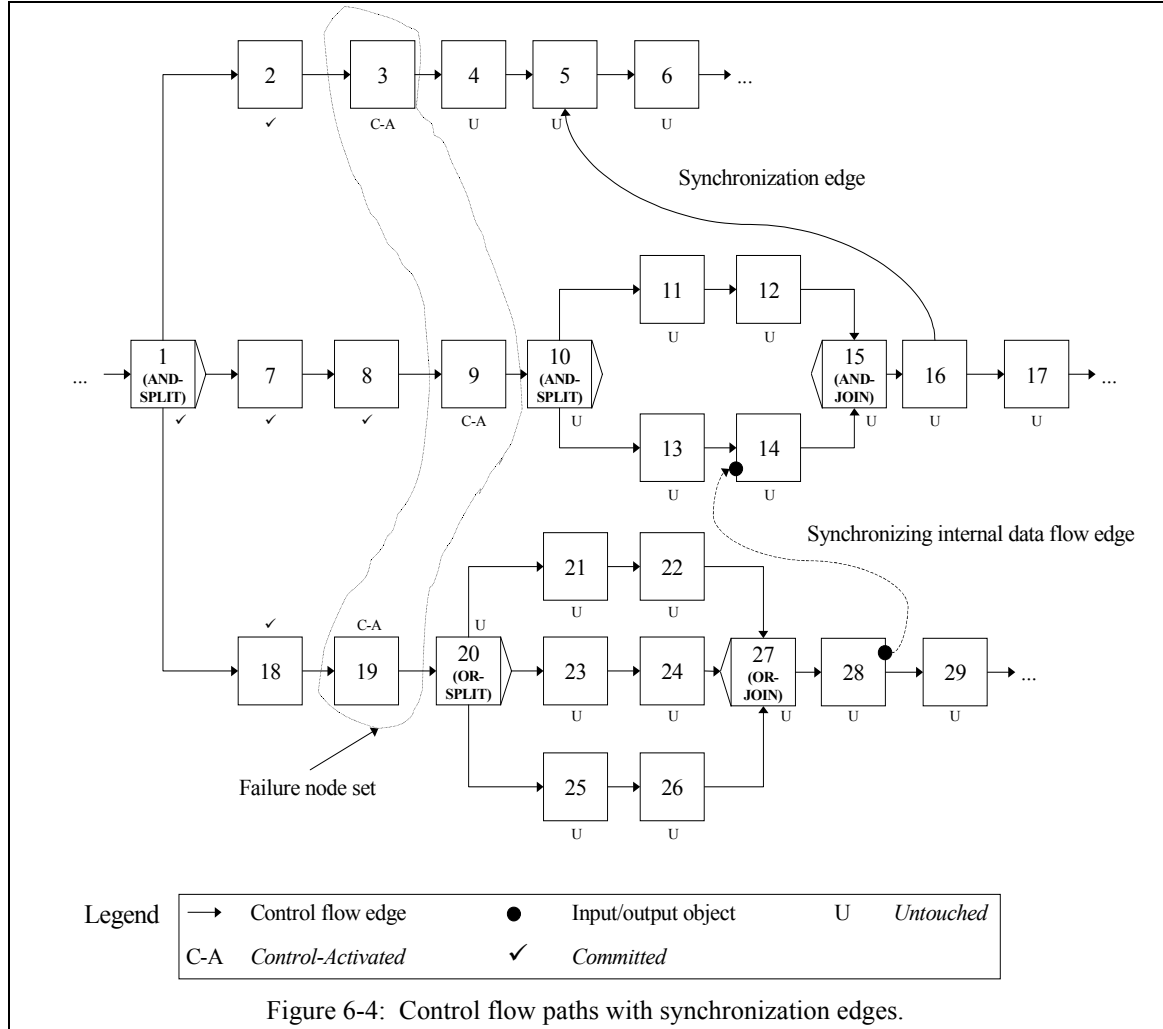
a control flow path as well. If there is more than one incoming or outgoing edge for a node, the edge has to be named explicitly instead of using the symbol \rightarrow , e.g., it has to be specified (22,27)27 \rightarrow 28 which means that the path starts with the edge (22,27) and ends with node 28.

In 6.5.1, we describe the estimation of control flow paths for which the execution duration is *not* influenced by synchronization edges or “synchronizing” internal data flow edges (as the one in Figure 6-2). In 6.5.2, we describe the estimation of control flow paths with such a synchronization. The results of 6.5.1 and 6.5.2 are then used to estimate entire workflows (6.5.3) and to determine during predictive adaptation the part P_{VT} that will be executed during a valid time VT (6.5.4).

For the workflow of Figure 6-4 we assume that all nodes of the failure node set just have been set to state *Control-Activated*, i.e., for the estimation of paths starting at a node n of the failure node set the full duration of n has to be considered. If this cannot be assumed (e.g., if node n already is in state *Active*) the duration during which n already has been executed has to be subtracted from the estimation value estimating the execution duration of n .

6.5.1 Control Flow Paths Without Synchronization

The estimation of control flow paths without synchronization can directly be derived from the estimation mechanisms described in sections 6.4.1–6.4.5. This is described best by an example for



which we use the path

$$p = 19 \rightarrow 20 \rightarrow 25 \rightarrow 26 \rightarrow 27 \rightarrow 28 \rightarrow 29$$

in Figure 6-4: First, the subpath until the last node before the next control node is determined. In Figure 6-4 this is the subpath $19 \rightarrow$. As this subpath is a sequence of activity or communication nodes, the estimation mechanisms of 6.4.1 can be used for it. Then, as the first node of the remaining sequence is an OR-SPLIT node (node 20), it has to be predicted by the mechanisms of 6.4.3 which of the remaining nodes of p will be executed at all. Let us assume that it can be predicted that the nodes 25 and 26 of our path p will be executed. Thus, the average duration of p to be assumed is

$$dur\text{-}av(p) = dur\text{-}av(19 \rightarrow) + dur\text{-}av(b) + dur\text{-}av(\rightarrow 28 \rightarrow 29) \quad (xliv)$$

where b is the block opened by node 20 and closed by node 27. Note that the whole duration of b has to be considered, i.e., $dur\text{-}av(b)$, and not only the duration of the subpath $20 \rightarrow 25 \rightarrow 26 \rightarrow 27$. This is because according to the execution model of an OR-SPLIT/OR-JOIN block (see 5.4.4.2) node 28 can be set to state *Control-Activated* only *after* all executed paths of b have been completed. The minimal and maximal duration of p is estimated analogously to (xliv).

If it is derived that nodes 25 and 26 will *not* be executed, this means that for p only the subpaths $19 \rightarrow 20$ and $27 \rightarrow 28 \rightarrow 29$ will be executed. For these subpaths, the estimation mechanisms of 6.3.1 and 6.4.1 are sufficient.

All other path constellations, i.e., paths with AND-SPLIT, AND-JOIN, LOOP-START, LOOP-END, START, or END nodes are estimated analogously by combining the estimation mechanisms of sections 6.4.1–6.4.5 according to the type of control nodes appearing in the path.

6.5.2 Control Flow Paths With Synchronization

We now describe the estimation of control flow paths with a synchronization, i.e., of control flow paths which consist of at least one node being a target of a synchronization or "synchronizing" internal data flow edge e . For example, in Figure 6-4 this is the case for the paths

$$p = 3 \rightarrow 4 \rightarrow 5 \rightarrow 6 \quad \text{and} \quad p' = 9 \rightarrow 10 \rightarrow 13 \rightarrow 14 \rightarrow 15 \rightarrow 16 \quad (xlv)$$

as node 5 is a target of the synchronization edge leading from node 16 to node 5, and as node 14 needs an output object from node 28. The edge (16,5) resp. the internal data flow edge leading from node 28 to node 14 may delay the execution duration of p resp. p' as their source nodes (i.e., nodes 16 and 28) are *not* predecessor nodes of their target nodes (i.e., of nodes 5 and 14). Thus, the estimation mechanisms described in 6.5.1 are not sufficient for such paths with synchronization.

We restrict our considerations to synchronization edges with the structure

$$((x, \textit{Committed}), (y, \textit{Control-Activated})) \quad (xlvi)$$

i.e., edges specifying that y cannot be set to state *Control-Activated* before x has not been set to state *Committed*. Thus we can only write (x,y) instead of the expression in (xlvi). The handling of synchronization edges using node states different from those in (xlvi) and of "synchronizing" internal data flow edges is analogously.

To describe the estimation of a control flow path p where at least one node is a target of a synchronization edge (x,y) we can assume without loss of generality that y is the *first* node of p which is a target of such a synchronization edge. To illustrate our description we use the sample path p in (xlv), i.e., $p = 3 \rightarrow 4 \rightarrow 5 \rightarrow 6$, and $(x,y) = (16,5)$.

The execution duration of p is estimated as follows: First, two paths p_y and p_x are to be determined which fulfill the following conditions:

- a) p_y is the maximal subpath of p , that leads from the failure node set to y , but does *not* contain y itself.
- b) p_x leads from the failure node set to x , ends with x , and can be assumed to be executed entirely.

For example, in Figure 6-4 this is the case for the paths

$$p_y = 3 \rightarrow 4 \rightarrow \quad \text{and} \quad p_x = 9 \rightarrow 10 \rightarrow 13 \rightarrow 14 \rightarrow 15 \rightarrow 16 \quad (xlvii)$$

Condition a) means that the maximal subpath of p is determined that starts at the failure node set and that is not yet influenced by (x,y) . Condition b) means that a path is determined that has to be executed to reach and execute x . If *no* path p_x exists meeting condition b), this means that at estimation time it has to be assumed that x will not be executed at all. For example, in Figure 6-4 this would be the case for a synchronization edge (21,5) if it would be determined that the path $19 \rightarrow 20 \rightarrow 21$ would not be executed as the branching condition between node 20 and 21 would be evaluated to FALSE. According to the execution semantics of a synchronization edge (5.4.2.1), y can then be set to state *Control-Activated* without having to wait for the commitment of (x,y) .

If however a path p_x exists meeting condition b), this means that y can only be set to state *Control-Activated* after both p_y and p_x have been executed. Thus, the average execution of p is

$$dur-av(p) = \max\{dur-av(p_y), dur-av(p_x)\} + dur-av(p - p_y) \quad (xlviii)$$

where $p - p_y$ denotes the remaining subpath of p after p_y , e.g., $p - p_y = 5 \rightarrow 6$ in Figure 6-4. The maximal and minimal execution duration of p is estimated analogously.

6.5.3 Entire Workflows

As a direct result of sections 6.5.1 and 6.5.2, we can now estimate the duration of an entire workflow. First, we have to predictively resolve all conditions at OR-SPLIT nodes to determine those paths starting at the START node that will be executed. Then, all control flow paths up to the END node have to be estimated, and the maximum of their average, maximal or minimal execution duration then is the average, maximal resp. minimal duration to be assumed for the workflow execution. As a consequence, we retrospectively can use *complex* activity definitions (which assign a whole workflow definition to an activity node) in the formulas in sections 6.4–6.5 as well.

It is clear that the duration estimation of an entire workflow usually will not be possible for all parts, as it is unlikely that all conditions at OR-SPLIT nodes can be predictively resolved. For these areas that cannot be estimated as conditions cannot be resolved predictively, AGENTWORK shifts to reactive adaptation.

6.5.4 Control Flow Path Estimation during Predictive Adaptations

What remains is to describe how the estimation of control flow paths is used during predictive adaptation, in particular how to determine that workflow part P_{VT} of a workflow instance I that is assumed to be executed during the valid time interval VT of a triggered control action ca . This can

now be described in a compact manner:

- First, for every node from the failure node set $FNS_{l, ca}$, all control flow paths are determined that may be executed.
- Second, for each of these paths its execution duration is estimated by average case estimation. In particular, at OR-SPLIT nodes it is tried to resolve conditions of the path predictively. The estimation of a path is terminated when VT is “consumed“ by the path (i.e., when it can be assumed that further nodes will not be executed anymore during VT), or when due to unresolvable conditions at OR-SPLIT nodes it cannot be derived for further nodes of the path whether they will be executed during VT or not.
- P_{VT} then consists of all node and edge executions that can be assumed to take place during VT due to the above path estimations. This part P_{VT} then is adapted predictively to satisfy the control action ca .
- All nodes for which it cannot be derived whether they will be executed during VT or not are handled reactively.

6.6 Related Work

Workflow estimation has been recently addressed by several authors, in particular to handle temporal constraints assigned to workflows (e.g., [SON & KIM 2001, DADAM ET AL. 2000⁵, EDER ET AL. 1999 A]). We first discuss the network planning technique [EISELT & FRAJER 1977], as many workflow estimation approaches are based on this general technique. Second, we discuss representative and specific estimation approaches from the field of workflow management.

Network Planning Technique: The network planning technique (NPT) is a method originating from the field of operations research [TAHA 1982]. It is based on graph theory and supports the definition, analysis, and control of projects and processes, whereby time, cost, resources, and other influential factors can be taken into consideration. For example, NPT allows to define the order of project activities and their durations and to analyze project definitions w.r.t. the question whether deadlines can be met. Two principal classes can be defined: In *deterministic network planning* [TAHA 1982] both the activity duration and the activity structure are considered as determinable. In *stochastic network planning* [DADUNA 2001, TOMII ET AL. 1999] it is assumed that a parameter such as an activity duration cannot be uniquely specified, but has to be handled as an independent random variable based on some probability distribution. In this thesis, we do not consider such stochastic planning techniques further, as AGENTWORK addresses applications where it can be assumed that execution durations are determinable, either by duration estimations at definition time or by duration measurements at execution time.

5. The estimation approach of the ADEPT_{FLEX} system [DADAM ET AL. 2000] which is used to determine the effects a dynamic adaptation might have for temporal constraints assigned to a workflow has been already discussed in Chapter 2 (*Related Work*) in Section 2.4.1.

Concerning *deterministic* network planning, a broad range of variants exist. For example, the *critical path method* (CPM) is an algorithm for finding the longest (i.e., “critical”) path(s) through a network [PHILIPOSE 1986, WIEST & LEVY 1977]. It calculates early start and early finish times for each activity in a forward pass through the network, while late start times, late finish times, and slack values are calculated in a backward pass. These times are matched to detect paths being critical for a timely project success. However, CPM has some limitations. For example, to model all the precedence relations among the activities of a project network, numerous *dummy activities* typically have to be introduced. Furthermore, only *minimal* time lags between activities can be modeled. In the AGENTWORK context, especially the latter limitation is not acceptable, as for example it must be possible to specify the maximal lag between two chemotherapy activities. To overcome the limitations of CPM, the so-called *metra-potential method* (MPM) has been developed [NEUMANN & MORLOCK 1993]. It uses two types of relations between the starting points of two activities, one for the minimal time lag and one for the maximal time lag. Furthermore, MPM considers limited renewable resources (e.g., machines) and overlapping operations. However, a limitation of MPM is that already for simple resource-constrained scheduling problems with arbitrary time lags the question whether or not a feasible schedule exists is NP-complete [BRUCKER ET AL. 1999].

Workflow Estimation Approaches: Several authors have recently dealt with workflows estimation [SON ET AL. 2001, SON & KIM 2001, DADAM ET AL. 2000, EDER ET AL. 1999 A, MARJANOVIC & ORLOWSKA 1999, KAFEZA & KARLPALEM 1999, ADAM ET AL. 1998]. For example, [EDER ET AL. 1999 A] allow to assign explicit time constraints to a workflow at workflow definition time, such as that the invitation for a meeting has to be mailed to the participants at least one week before, or that a final patent filing has to be done within a certain time period after the preliminary filing. Furthermore, it is assumed that for every activity a deterministic duration has been assigned. On the basis of the above-mentioned CPM, it is then checked at workflow definition time whether for a given workflow definition there exists an execution schedule that does not violate any time constraints. The result is a so-called *timed activity graph* that includes deadline ranges for each activity. At workflow instantiation time, this timed activity graph is extended by including the deadlines and date characteristics given when the workflow is started. At workflow execution time, the timed graph is dynamically recomputed for the remaining activities to derive the activity completion times of the remaining workflow part and thus to monitor that the remaining time constraints are satisfied.

In [SON ET AL. 2001, SON & KIM 2001] the authors present a method to find out the critical path with the longest average execution time in a workflow. For this, an extension of CPM is provided which is called the innermost control structure first method (ICSF). This method first determines a sub-critical path with the longest average execution time in each control block, and then combines these sub-critical paths. When a sub-critical path is determined for each control block, the longest execution path is selected from the innermost control structure to the outermost control structure. Because the critical path directly affects the overall execution time of a workflow, additional workflow processing capacities (i.e., workflow servers) can be dynamically allocated to maximize the number of workflow instances that satisfy specified deadlines.

Based on MPM, [MARJANOVIC & ORLOWSKA 1999] allow to assign a minimal and maximal duration to every activity, and provide algorithms based on the *shortest path partitioning algorithm* [EVANS & EDWARD 1992] to estimate the shortest and longest duration of concurrent, conditional, and synchronized control flow structures.

However, beside the general limitations of CPM and MPM mentioned above, there are several specific limitations of these workflow estimation approaches:

- First, only duration estimations obtained at workflow definition time are supported. In particular, measurements at execution time are not considered to obtain more realistic estimation values than it typically will be possible at workflow definition time. Furthermore, the authors do not provide any mechanisms to *group* estimation values by several dimensions such as user types or programs, or by day times. As the quality of the estimation values is the major precondition for high quality workflow estimation, this must be viewed as not sufficient.
- Second, the duration of data flow edges is neglected. As discussed in 6.2.2, it has to be assumed that the executions of *external* data flow edges (i.e., edges accessing some external data sources) not always have a negligible duration. This holds especially for edges requesting data from a workflow user, e.g., edges requesting data needed for a condition evaluation.
- Third, no attempt is made to predictively resolve conditions at a conditional branching or in a loop. We have described in Section 6.4.3 that under the circumstance that the data needed for the condition evaluation is already available, this may be possible and would increase the quality of workflow estimation significantly.

For further recent related work, which does not specifically address temporal aspects for workflow management but more generally for project management and job scheduling, we refer to [KERZNER 2001, BLAZEWICZ ET AL. 2001, NAYLOR 1995].

6.7 Summary and Discussion

In this chapter, we described how AGENTWORK estimates the execution duration of workflows. For this, we first described how the durations of *edge* resp. *node* executions are estimated. One characteristic has been that AGENTWORK supports two principal possibilities to obtain estimation values, namely duration estimations at definition time and duration measurements at execution time. By combining these two possibilities, realistic estimation values for edges and node can be obtained and thus a high estimation quality can be achieved. For example, one can use duration estimations specified at workflow definition time for the first phase of an AGENTWORK installation and then can continuously refine them by temporal measurements performed during the operational phase of the system. Furthermore, to achieve fine-grained context-dependent estimations, estimates can be grouped by different dimensions such as the activity types or the needed user types and programs.

Then, we have described how more complex workflow parts such as sequences of activity and communication nodes, workflow blocks, and arbitrary control flow paths can be estimated. A

major characteristic of this has been that AGENTWORK provides mechanisms to predictively resolve conditional branching to better predict which workflow parts will be executed during which temporal interval. Finally, for all estimations AGENTWORK provides different strategies, namely worst case estimation, best case estimation, and average case estimation. The strategy of average case estimation is the default strategy used by AGENTWORK during predictive adaptation, as this strategy keeps the situations where further adaptations have to be made to satisfy a control action in balance with those situations where adaptations have to be taken back. As we will see in Chapter 10 (*Handling Control Flow Failures for Cooperating Workflows*), the other two strategies will be used to derive what an adaptation means in the best and the worst case for the temporal agreements specified between two cooperation partners.

There are several limitations and disadvantages that have to be considered. First of all, duration measurements at execution time typically will provide realistic estimation values only after workflow execution durations have been measured for quite a long time, e.g., for several months. Thus, during the first phase of an installation and operational usage of a workflow application based on AGENTWORK, only estimation values obtained during workflow definition time can be used. As such a first phase is the most critical one concerning user acceptance, this means that like for other approaches [DADAM ET AL. 2000, EDER ET AL. 1999 A, MARJANOVIC & ORLOWSKA 1999], the success of AGENTWORK significantly depends on high-quality estimation values obtained at workflow definition time. This may be critical, as for some workflow applications it may be not easy to specify realistic estimation values obtained at workflow definition time.

Furthermore, the attempt to predictively resolve a condition increases the complexity of a workflow monitoring that has to be performed when the estimated (and adapted) workflow is continued. This is because it cannot be excluded that branching conditions are predicted wrong, so that different paths are executed than originally assumed. Thus, a monitoring does not only have to compare the actual execution durations of nodes and edges with the estimated durations (as already sketched in Section 3.4.5), but also has to compare the predicted results of a condition evaluation with the actual evaluation results.

Last not least, only an *evaluation* of the described estimation algorithms *under real-world conditions*, such as with workflows and patients of the HEMATOWORK system, will allow for a final decision about the quality of the AGENTWORK estimation approach. As the technical and organizational preconditions for such a real-world evaluation have not been established for HEMATOWORK when this thesis has been completed, the author did not yet had the possibility to perform such a real-world evaluation. Rather, only a few “laboratory” tests with medical workflows running for fictitious patients have been done to evaluate the estimation algorithms [GREINER 2000]. Nevertheless, these tests have shown that under “laboratory” conditions the estimation algorithms are working well.

This chapter describes the control actions that are provided by AGENTWORK to handle control flow failures. As already sketched in Chapter 3, AGENTWORK uses two main classification criteria for control actions: *Global* control actions deal with a workflow as a whole, while *local* control actions deal only with *some* activities of a workflow. Orthogonal to this, a control action may be *case*-related or *resource*-related. Case-related control actions are triggered when an event occurs to a *case*. They state what has to be done with a workflow or some of its activities from the case point of view. Resource-related control actions are triggered when an event occurs to a *resource*. They state what has to be done with a workflow or some of its activities from the resource point of view.

The chapter is organized as follows: Sections 7.1 and 7.2 list the *global* resp. *local* control actions supported by AGENTWORK. Section 7.3 introduces some useful conventions concerning a control action's valid time. These conventions are made to facilitate the failure handling process in the following chapters. Section 7.4 describes how triggered control actions are processed further. In particular, this section describes how it is determined which workflows or activities are *affected* by triggered control actions. In Section 7.5, we discuss integrity aspects of rules containing control actions. Section 7.6 describes how AGENTWORK copes with dynamic control action dependencies that may occur when several control actions are triggered simultaneously. The chapter concludes with a summary and discussion in Section 7.7.

The following notational conventions will be frequently used in the subsequent sections: C will denote an object of class *Case*, and R will be of type *Obj-Patt*<*Resource*>, i.e., a resource pattern (according to 4.2.1.5). If new data for a case or a resource is inserted into an extension, and if *new* references this new data, then C_{new} resp. R_{new} denotes the *Case* resp. *Resource* instance described by

Global Control Action (with valid time <i>VT</i>)	Meaning
Case-Related:	
<i>abort(W,C)</i>	Abort any workflow based on <i>W</i> and running (exclusively) for case <i>C</i> . <i>VT</i> has to be a single point in time specifying when the workflow <i>W</i> shall be aborted.
<i>suspend(W,C)</i>	For the time interval specified by <i>VT</i> , suspend any workflow based on <i>W</i> and running for case <i>C</i> . Any node of the remaining control flow must not be executed before the time specified by <i>VT</i> has elapsed. <i>VT</i> always has to be a valid time <i>interval</i> .
Resource-Related:	
<i>abort-workflows-of(R)</i>	Abort any workflow for which a resource described by <i>R</i> is needed for the remaining control flow. The restrictions concerning <i>VT</i> are the same as for <i>abort</i> .
<i>suspend-workflows-of(R)</i>	For the time interval specified by <i>VT</i> , suspend any workflow for which a resource described by <i>R</i> is needed for the remaining control flow. The remarks concerning <i>VT</i> are the same as for <i>suspend(W,C)</i> .
Legend <i>W</i> := workflow definition according to 5.3.9	

Table 7-1: Global control actions.

this data. For example, if *infection-findings* is an extension for objects of class *Infection-Finding*, then C_{new} refers to the patient whose infection is described by a new *Infection-Finding* object inserted into *infection-findings*. *VT* always will denote the valid time of a control action.

7.1 Global Control Actions

Table 7-1 lists the global case- and resource-related control actions supported by AGENTWORK. *Case*-related global control actions are only allowed for workflows which are executed for only one case during their life span (and for which this case consequently has to be known already at workflow initialization time). This restriction is necessary, as it would not make much sense to abort or suspend a workflow executing activities also for other cases than the one to which the failure triggering event happened. In contrast to this, *resource*-related global control actions can be used for any workflow regardless for how many cases it is executed or how many other resources are needed to execute it. This is because they express that because of an event occurring to a resource a workflow needing this resource cannot be continued, regardless for how many cases this workflow is executed. For example, when an important diagnostic device gets broken and no substitution device is available, it does not make sense to continue a diagnostic workflow using this device. If only some activities shall be removed from the workflow due to some resource event, local resource-related control actions have to be used (see 7.2).

We give two failure rule examples using global control actions:

1. Let W be some cancer chemotherapy workflow definition such as the one in Figure 1-1. Let furthermore *infection-findings* be an extension for objects of class *Infection-Finding* with a string attribute *type* and an float attribute *fever*. Then the rule

WHEN *INSERT ON infection-findings* (i)
WITH *new.type = "Infection of Upper Respiratory Tract" AND new.fever > 39°*
THEN *suspend(W, C_{new}) VALID-TIME [now, now + (2, week)]*

states that whenever a patient (referenced by C_{new}) has an infection of the upper respiratory tract with fever higher than 39 Celsius degree, any chemotherapy workflow based on W and running for C_{new} has to be suspended for two weeks (assuming that the patient will recover from this infection during this time). Alternatively, the valid time of *suspend* in (i) could also be conditional in the sense “until infection is over”.

2. Let W be some workflow definition for a nuclear spin tomography examination, and let $R := \text{Computer-Tomograph}[]$ be a resource pattern. Then the rule

WHEN *instances-not-available(R)*¹
THEN *abort-workflows-of(R) VALID-TIME now*

states that when there is no nuclear spin tomograph available, workflows for which a computer tomograph is needed have to be aborted immediately.

7.2 Local Control Actions

Local control actions mean that a workflow in principle can be continued but has to be adapted locally due to a failure event. In addition to the distinction between case- and resource-related control actions, AGENTWORK uses another classification criteria for local control actions which is useful for control action processing. So-called *non-additive* control actions refer to activities that are already existent in a workflow and may have to be dropped, replaced, or postponed. They are called *non-additive* as they do not add new activity nodes to a workflow. In contrast to this, *additive* control actions insert new activity nodes to a workflow. The reason for distinguishing these two classes is the following: Non-additive control actions “only” require that currently executed workflows are scanned whether they contain activities in their remaining control flow that either

- first match the activity pattern of the control action and second are executed for the case referenced by the control action (for case-related control actions), or that
- match the resource pattern of the control action (for resource-related control actions).

1. Please recall from 4.2.2.1 that the predicate *instances-not-available(R)* returns TRUE when no *Resource* instance exists that matches R .

Local Control Action (with valid time VT)	Meaning
Case-Related:	
$drop(A, C)$	During VT , any A -node execution for C has to be dropped.
$replace(A, B, C)$	During VT , any A -node execution for C has to be replaced by a B -node execution.
$postpone(A, d, C)$	During VT , any A -node execution for C has to be postponed by distance d (relative to its control flow position at the point in time the control action has been triggered).
$review(A, C)$	During VT , any A -node execution for C has to be reviewed by an authorized user.
$change-value(A, p, f, C)$	During VT , for any A -node execution for C the value of p shall be changed according to the function f . Parameter p is an object path of A (e.g., an attribute of A), and f a function defined over the domain of p .
Resource-Related:	
$drop-activities-of(R)$	Any R -node execution during VT has to be dropped.
$postpone-activities-of(R, d)$	Any R -node execution during VT has to be postponed by distance d (relative to its control flow position at the point in time when the control action has been triggered).

Table 7-2: Non-additive local control actions.

In contrast to this, additive control actions require that first a workflow into which new activity nodes can be inserted has to be *identified* or *generated*. Second, they require that an *appropriate insertion point* within such a workflow is determined.

In the following, A and B denote activity patterns according to 4.2.1.5. Parameter d is a temporal distance which is conform to the used temporal frame (4.3.3). As in Chapter 3, we call a node for which the assigned activity definition matches an activity pattern A an A -node. Analogously, we call a node for which execution a resource described by R is needed a R -node.

7.2.1 Non-Additive Local Control Actions

Table 7-2 lists the non-additive control actions supported by AGENTWORK. We give two failure rule examples using this type of control actions:

1. An example for a rule with a case-related non-additive control action is the following. For

$A := \text{Drug-Administration}[\text{drug} = \text{"ERYTHROMYCIN"}]$ and
 $B := \text{Drug-Administration}[\text{drug} = \text{"DOXYCYCLIN"}]$

the rule

WHEN *INSERT ON* *hemato-findings* (ii)
WITH *new.parameter = Leukocyte-Count AND new.value < 1000*
THEN *replace(A, B, C_{new}) VALID-TIME [now, now + (1, week)]*

states that whenever a patient has a leukocyte count less than 1000, all ERYTHROMYCIN administrations have to be replaced by DOXYCYCLIN administrations during the next week.

2. An example for a rule with a resource-related non-additive control action is (for $X := \text{Physician}[\text{degree} = \text{Senior}, \text{speciality} = \text{"Oncology"}]$):

WHEN instances-not-available(X) VALID-TIME [now, now + (n, day)] (iii)
THEN drop-activities-of(X) VALID-TIME [now, now + (n, day)]

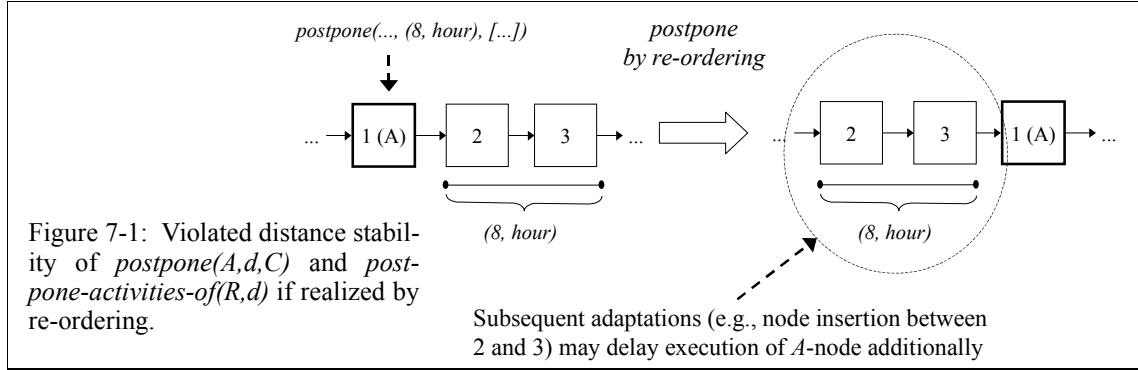
This rule states that when there is no senior specialist available for the next n days (e.g., due to illness or absence), all nodes for which such a senior specialist is needed have to be dropped during these n days.

We now describe specific aspects of control actions listed in Table 7-2:

Control actions *postpone(A,d,C)* and *postpone-activities-of(R,d)*:

The motivation for these two control actions is that it sometimes is sufficient not to suspend a whole workflow by the global control action *suspend(W,C)*, but to postpone only some of its activities. For example, concerning the case-related control action *postpone(A,d,C)*, imaging a workflow supporting the treatment of a patient. It then may be sufficient to postpone only the patient's drug administrations while the diagnostic activities for this patient can be executed according to the original workflow definition. Analogous examples can be found for the resource-related control action *postpone-activities-of(R,d)*. For example, difficult medical activities requiring the presence of the senior physician may have to be postponed when no senior physician is available – as shown in rule (iii) –, while other activities of a workflow only requiring an assistant physician may still be executed.

A principal question is whether it should be allowed that subsequent adaptations influence an activity postponement which has been induced by *postpone(A,d,C)* or *postpone-activities-of(R,d)*. For example, let us assume that the execution of an A -node 1 shall be postponed by 8 hours (Figure 7-1). If the estimated execution duration of the two successor nodes 2 and 3 is 8 hours, then this could be realized by re-ordering the node sequence, i.e., to insert node 1 behind node 3. However, then a subsequent adaptation such as inserting a node between 2 and 3 may have the consequence that the A -node will be postponed significantly more than 8 hours. It is the assumption of AGENTWORK that such side-effects of subsequent adaptations should be avoided, i.e., that the relative temporal *distance* between the old and the new position of a postponed activity node may not be changed by subsequent adaptations (distance stability). For example, if a physician postpones



drug administrations this usually means that additional diagnostic activities should not delay the postponed drug administrations additionally. In contrast to this, adaptations that do not change the relative distance between the old and the new position of an postponed activity should be allowed. This distance stability constraint of $\text{postpone}(A, d, C)$ and $\text{postpone-activities-of}(R, d)$ will influence the design of a control flow operator transforming a node postponement into structural workflow adaptations significantly as we will see in Chapter 8 (*Structural Adaptation Operators*).

Furthermore, it could be asked why AGENTWORK does not support a control action that “brings forward” an activity node to achieve an earlier execution of the node. The reason is that such a “bring forward” control action is difficult to realize with a reactive adaptation strategy: A node n that would be affected by such a control action would have to be handled not later than the point in time t' with $t' - d$, where t' is the point in time for which n originally has been scheduled for execution, and d the duration by which n shall be executed earlier. This means that starting from node n those predecessor nodes have to be determined that are executed earlier than n by distance d , to move n to this predecessor nodes. For this, temporal estimation is needed, or – in other words – only predictive adaptation is possible. However, as temporal estimation may not always be possible, this means that there may be constellations where no adaptation strategy is applicable. To avoid this, such a “bring forward” control action is not supported.

Control action $\text{review}(A, C)$:

A $\text{review}(A, C)$ control action is used when there is not enough knowledge available to automatically decide whether an A -node is adequate or not for case C . For example, a laboratory value may be within a range for which it is not clear whether a drug such as ETOPOSID should be dropped or not (as ETOPOSID may influence the laboratory value in a negative way). Thus, the ETOPOSID administration has to be reviewed. When a $\text{review}(A, C)$ control action is triggered, the user is requested to specify for each A -node whether it still shall be executed according to the workflow definition, or whether it for example shall be dropped, replaced, or postponed.

Control action $\text{change-value}(A, p, f, C)$:

The $\text{change-value}(A, p, f, C)$ control action has been introduced as sometimes it is sufficient to

Control Action (with valid time <i>VT</i>)	Meaning
<i>add(A,C)</i>	One additional <i>A</i> -node for <i>C</i> has to be executed during <i>VT</i> .
<i>add-repetitively(A,d,C)</i>	Additional <i>A</i> -nodes executions have to be performed repetitively for <i>C</i> during <i>VT</i> . The duration between two <i>A</i> -node executions is specified by <i>d</i> .

Table 7-3: Additive local control actions.

dynamically change component values of an activity node (in particular attribute values) instead of dropping or replacing the whole activity node. A typical medical example is the dosage reduction due to some critical, however not severe drug toxicity. A rule example using this control action could be the following: For

- $A := \text{Drug-Administration}[\text{drug} = \text{"ETOPOSID"}]$,
- $f(x) := x * 0.5$,
- P being an object of class *Patient*, and
- $\text{critical-hemato-finding}(P)$ being the predicate introduced in Section 4.2.2,

we can define the rule

$\text{WHEN critical-hemato-finding}(P) \text{ VALID-TIME } [\text{now} - (1, \text{week}), \text{now}]$ (iv)
 $\text{THEN change-value}(A, A.\text{dosage}, f, P) \text{ VALID-TIME } [\text{now}, \text{now} + (2, \text{week})]$

which states that whenever a patient has a critical hematological finding for at least one week, the dosage of all ETOPOSID administrations should be reduced by 50 percent for the next two weeks.

7.2.2 Additive Local Control Actions

Table 7-3 lists the additive local control actions supported by AGENTWORK. In contrast to non-additive control actions, AGENTWORK does support only case-related additive control actions, but no resource-related additive control actions. This is because it is difficult to find relevant examples where the absence of a staff member or the damage of a piece of equipment induces additional activities for the affected workflow.²

An example for a rule with an *add* control action is the following: For

$A := \text{Sonography}[\text{focus} = \text{"Heart"}]$ and

2. For a workflow supporting the repair of a broken piece of equipment this piece of equipment is not anymore a resource for this workflow but would play the role of a *case* for which the activities of this repair workflow are executed.

cardiological-findings being and extension for objects of the class
Cardiological-Finding[*type*: String, *degree*: Enum{SEVERE, CRITICAL, NORMAL}]

we can define the rule

(v)

WHEN *INSERT ON cardiological-findings*
 WITH *new.type* = “Inflammation of heart muscle” AND *new.degree* = SEVERE
 THEN *add(A, C_{new}) VALID-TIME [now, now + (2, day)]*.

This rule states that when a patient shows some severe inflammation of the heart muscle, a heart sonography should additionally be executed during the next two days. Alternatively, by using

(vi)

THEN *add-repetitively(A, (3, day), C_{new}) VALID-TIME [now, (2, week)]*

in rule (v) it can be specified that a heart sonography should be done every three days for the next two weeks.

We make the following remarks:

- Obviously, an *add-repetitively(A, d, C)* control action could also be expressed by a combination of several *add(A, C)* control actions. However, as the repetitive execution of activities occurs frequently at least in medical domains an extra control action has been introduced for this.
- For an *add(A, C)* control action, the end of *VT* cannot be described by a termination condition such as *Until normal-hemato-status(C)*. This is because it then would be unclear when a new activity node should be inserted into the control flow, as the point in time when the condition becomes true and therefore the end of *VT* is not known beforehand. In contrast to this, for an *add-repetitively(A, d, C)* control action the end of *VT* may also be specified by a termination condition. This is possible as the execution points in time of the new repetitive activity nodes are specified by the period *d*, and as the termination condition then simply states when this repetitive execution shall terminate.

7.2.3 Further Aspects

We now discuss some aspects being relevant for all local control actions. This includes the discussion whether local control actions shall postulate that affected nodes are executed *entirely* or only *started* during the valid time interval (7.2.3.1). Furthermore, we discuss why *case patterns* are not allowed in control actions (7.2.3.2), how so-called *deadline events* can be handled by control actions (7.2.3.3), and how the *user* may interact (7.2.3.4).

7.2.3.1 Entire Execution versus Execution Start during Valid Time Intervals

An open question is whether the term “*execution during a valid time*” used in Table 7-2 and Table 7-3 means that an activity node has to be executed entirely during the valid time or only has to be started (i.e., set to state *Active*) during the valid time. Concerning this question, AGENTWORK only postulates that the *start* of the execution of a node *n* has to occur during a valid time *VT*, if *n* shall

be affected by a non-additive control action, or shall be executed additionally due to an additive control action. It is not required that n is *entirely* executed during VT . The reason for this is that a reactive adaptation would not be possible if one would postulate that n is *entirely* executed during VT . For example, if a $drop(A, C)$ has been triggered, the question whether an A -node is executed entirely during VT and thus is affected by $drop(A, C)$ cannot be determined before the A -node has been set to state *Committed*. However, then it is too late to drop the node execution as the activity assigned to it has already been performed.

Typically, the fact that AGENTWORK only postulates the execution *start* during the valid time interval, affects only activity node executions at the “right end” of the interval, as valid time intervals usually are much longer than activity durations.

7.2.3.2 Case Instances and Resource Patterns

As events may affect both cases and resources, we have introduced case-related and resource-related control actions. However, one may ask why the case parameter C in a control action has to be a concrete *Case* instance (e.g., patient John Miller), while it may be a resource *pattern* for the parameter R . The answer is the following: While already an event affecting a single *Case* instance may cause a control flow failure, events occurring to resources typically cause control flow failures only when they affect *all* resource instances meeting some pattern, such as *all* instances of a computer tomograph. This is because then no proxy is available anymore so that activity nodes for which such a resource is needed cannot be executed. This has also been reflected by the *instances-not-available* predicate used for example in rule (iii).

Of course, one could argue that using *Case* patterns instead of *Case* instances in control actions would give more flexibility to failure handling. For example, let us assume that the knowledge becomes available that for some disease D a drug should not be used anymore in general, as this drug appears to be too dangerous for treating D . Then, one could think of a *drop* control action version that states that activity nodes administering this drug and executed for *any* patient suffering from D should be dropped. However, as indicated by this example, this is more a matter of adapting workflow *definitions* at built time (e.g., workflow definitions dealing with disease D), instead of adapting workflow instances at run time. This is because not a single case instance but a case *collection* is affected by the event that a drug appears to be too dangerous. Thus, using case patterns would mix up the problems of adapting workflow definitions (being a matter of *schema evolution*; see 2.4.1), and of adapting workflow instances. Therefore, AGENTWORK does not support case patterns in failure handling rules.

7.2.3.3 Control Actions and Deadline Events

In this section we show that the event and control action model of AGENTWORK is also able to handle so-called *deadline events* that play an important role in many workflow application. Principally, a deadline event occurs if it becomes clear that some given deadline cannot be met anymore (such as a deadline for producing a product for some customer). In workflow terms, this typically means that a workflow (part) which has been assumed to be finished until this deadline cannot be completed anymore until the deadline. As a consequence, dynamic adaptations may have to be per-

formed to meet the deadline, such as dropping activity nodes not being mandatory. This can be expressed by rules such as follows (W denotes some workflow definition supporting the preparation of a product, C the customer for whom this product has to be produced, and dl_C the deadline point in time that has to be met for C):

WHEN $\text{left-time-to}(dl_C) \leq (n, \text{day})$ *AND* $\text{needed-time-to-complete-workflow}(W) > (n, \text{day})$
THEN $\text{drop}(A, C)$ *VALID-TIME* [now, dl_C]

This rule states that whenever the time left until the deadline is only n days anymore (or less), but the time needed to execute the remaining workflow is more than n days, all A -nodes have to be dropped from the workflow from now until the deadline. The function $\text{needed-time-to-complete-workflow}(W)$ evaluates the duration needed to complete the workflow. The estimation algorithms needed for this have been described in Chapter 6.

7.2.3.4 User Interaction

Triggered control actions may have massive effects on a workflow, such as dropping drug administration nodes for a patient. Thus, there should be human interaction checking whether the triggered control actions are really appropriate to handle a control flow failure. Therefore, in AGENTWORK an authorized user has the possibility for any triggered control action to reject it, or to change its parameter values or its valid time interval, or to refine it by subclassing the used activity, resource or case classes. For example, let us assume that an $\text{add}(A, C)$ control action has been triggered with

$A := \text{Drug-Administration}[\text{drug} = \text{"DOXYCYCLIN"}, \text{dosage} = 200, \text{unit} = \text{mg},$ (vii)
 $\text{type} = \text{tablet}]$

specifying that the antibiotic drug DOXYCYCLIN should be given orally with a dosage of 200 mg. Then, the physician may want change A to A' as follows

$A' := \text{Drug-Administration}[\text{drug} = \text{"DOXYCYCLIN"}, \text{dosage} = 200, \text{unit} = \text{mg},$ (viii)
 $\text{type} = \text{infusion}]$

to specify that DOXYCYCLIN dosage that shall be given as infusion instead orally.

The question which users are “authorized” and which not depends on the particular workflow application, and therefore is considered as a matter of implementation (see Chapter 11). Thus, it is not discussed in detail in this chapter.

7.3 Valid Time Conventions for Control Actions

We now discuss some useful conventions concerning the valid times of control actions in active rules. As a prerequisite, we unify the notation of valid times by using the *VALID-TIME* keyword for fixed and conditional valid times, e.g., by writing

THEN $\text{drop}(\text{ETOPOSID}, P)$ *VALID-TIME* [$\text{now}, \text{Unless normal-hemato-status}(P)$]

instead of

THEN drop(ETOPOSID, P) Unless normal-hemato-status(P)

in the following. By this we can avoid syntactical case distinctions when specifying failure rules.

In ACTIVETFL, the following two conventions concerning the structure of a control action's valid time interval are made:

Restriction 1: **Beginning at now**. The valid time always has to start *now*, i.e., the point in time the failure rule has been triggered. In particular, this implies that the valid time of *abort(W, C)* and *abort-workflows-of(R)* always has to be the point in time *now*.

Restriction 2: **Coherence**. The valid time must not consist of several *unconnected* parts. For example, a valid time such as

$$[now, now + (3, day)] \cup [now + (5, day), now + (7, day)]$$

is not allowed.

These two conventions facilitate failure handling significantly. This is mainly because workflow estimation and adaptation then only have to cope with a workflow part which starts at the nodes that are currently executed or prepared for execution at the point in time of the failure (convention 1), and which is *connected* (convention 2).

However, as we have to assume that in many domains there will be failure rules which do not fulfill these conventions, we show how AGENTWORK transforms rules with arbitrary control action valid time intervals into rules meeting conventions 1 and 2. In the following, *E* describes some event triggering a failure rule, t_E the point in time when *E* occurred, *ca* some control action, and d_1, d_2, d_3 some temporal durations. The *WITH* part is omitted in the following examples as it is irrelevant.

First, let us assume that convention 1 is not met, i.e.

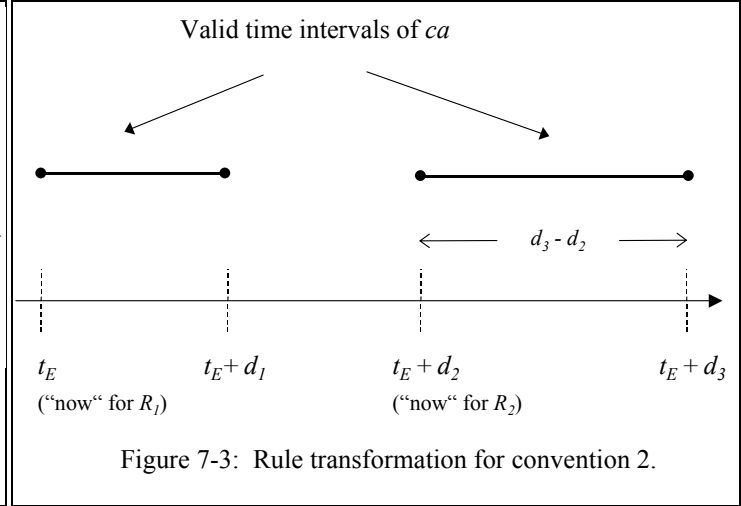
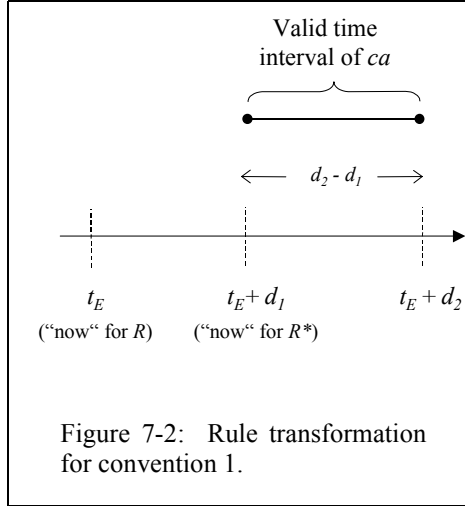
$$R: \text{WHEN } E \quad \text{THEN } ca \text{ VALID-TIME } [now + d_1, now + d_2] \quad (ix)$$

with $0 < d_1 < d_2$. This rule can be transformed into a rule meeting convention 1 (Figure 7-2):

$$R^*: \text{WHEN } \text{time-point-after}(E, d_1) \text{ THEN } ca \text{ VALID-TIME } [now, now + d_2 - d_1], \quad (x)$$

where *time-point-after*(*E*, *d*) is defined as the predicate that is true exactly at the point in time $t' = t_E + d$ in case *E* occurred, and false otherwise. Note that the point in time to which *now* refers is different in (ix) and (x), as *now* refers to the point in time when the particular rule is triggered.

The transformation of valid time intervals where convention 1 is not fulfilled and where the end is specified conditionally via *Until* or *Unless* is done analogously. A rule where convention 1 is not met because the *beginning* of the control action valid time is *conditional* can easily be transformed to a rule meeting convention 1 by moving the beginning condition to the *THEN-WITH* part of the rule.



Second, let us assume that convention 2 is not met. For this we assume that the valid time of a control action is split up into *two* intervals (the handling of more than two intervals is analogously), i.e.

$$R: \text{WHEN } E \text{ THEN } ca \text{ VALID-TIME } [now, now + d_1] \cup [now + d_2, now + d_3] \quad (xi)$$

with $0 < d_1 < d_2 < d_3$. Such a rule can be transformed into the two following rules (Figure 7-3):

$$R_1: \text{WHEN } E \text{ THEN } ca \text{ VALID-TIME } [now, now + d_1]$$

$$R_2: \text{WHEN time-point-after}(E, d_2) \text{ THEN } ca \text{ VALID-TIME } [now, now + d_3 - d_2].$$

R_1 is triggered when E occurs (i.e., at t_E), rule R_2 is triggered at $t_E + d_2$ (if E occurred). The situation that the end of the right interval in (xi) is not defined via $now + d_3$, but via *Until* or *Unless* is handled analogously (i.e., instead of using $now + d_3 - d_2$ the valid time in R_2 is terminated via *Until* or *Unless*).

Due to these two transformation mechanisms, we will assume only rules meeting conventions 1 and 2 for the following.

7.4 Control Action Processing

We now describe how triggered control actions are further processed. First, we introduce a useful definition, namely the *failure node set* which describes the execution stage of a workflow at the point in time of a control flow failure (7.4.1). Second, we describe how *global* control actions are processed (7.4.2). Third, we describe how *local* control actions are processed (7.4.3).

7.4.1 Failure Node Set

The failure node set describes at which execution stage a workflow is affected by a control flow failure.

Definition 7.1: Failure Node Set

Let ca be an arbitrary control action, and let I be a currently executed workflow instance. Then, we define as the failure node set $FNS_{I,ca}$ the set of nodes for which each member n fulfills the following two conditions at the point in time when ca has been triggered:

1. n is not in state *Committed* or *Unreachable*.
2. All direct predecessor nodes of n not being in state *Unreachable* are in state *Committed*.

Condition 1 means that n belongs to the *remaining* control flow that has still to be executed. Condition 2 means that n is currently executed (i.e., is in state *Active*), or will be executed *next*³. Thus, the failure node set $FNS_{I,ca}$ describes the execution “focus” of I at the point in time of a control flow failure. In particular, for local control actions that are handled by predictive adaptation, the failure node set serves as a description of the *beginning* of the workflow part that has to be estimated. If the triggering control action ca or the workflow instance I is clear from the context, we may omit the resp. index, i.e., we may write FNS_I , FNS_{ca} , or FNS .

7.4.2 Processing of Global Control Actions

7.4.2.1 Processing of $abort(W,C)$ and $abort-workflows-of(R)$

If an $abort(W,C)$ control action has been triggered, every workflow instance I based on the workflow definition W and executed exclusively for case C is aborted as described in 5.4.5 (i.e., the state transition sequence *Active* \rightarrow *Rolling-Back* \rightarrow *Aborted* is performed for I). Analogously, if an $abort-workflows-of(R)$ control action has been triggered, every workflow instance I for which at least one activity node of the remaining control flow needs a *Resource* instance matching the pattern R is aborted. Note that AGENTWORK does not support a *partial* rollback for aborting control actions. This would require that the aborting control actions could specify which workflow *part* should be rolled back, which is not supported at the moment.

7.4.2.2 Processing of $suspend(W,C)$ and $suspend-workflows-of(R)$

If a $suspend(W,C)$ control action has been triggered, every workflow instance I based on the workflow definition W and executed exclusively for case C is suspended. Analogously, if an $suspend-workflows-of(R)$ control action has been triggered, every workflow instance I is suspended for

3. The “unreachable” subcondition in condition 2 is necessary as n may not be an activity node but an OR-JOIN node, for which not all predecessor node have to be executed.

As it is difficult to automatically decide how to deal with such active nodes, AGENTWORK requests an authorized user how to cope with such nodes. This means, that for an instance I that has to be suspended, the user is requested for every node n of the failure node set $FNS_{I, ca}$ being in state *Active*, whether n shall commit or shall be aborted. As the question whether to commit or abort is a matter of single nodes, it may happen that for a workflow instance I one currently active path may commit w.r.t. its currently active node, while another path has to be rolled back to the last committed node. For example, let us assume that the workflow in Figure 7-4 supports a chemotherapy and has to be suspended due to some toxicity. Let us furthermore assume that the currently executed node 3 may administer a drug responsible for the toxicity and therefore has to be aborted immedi-

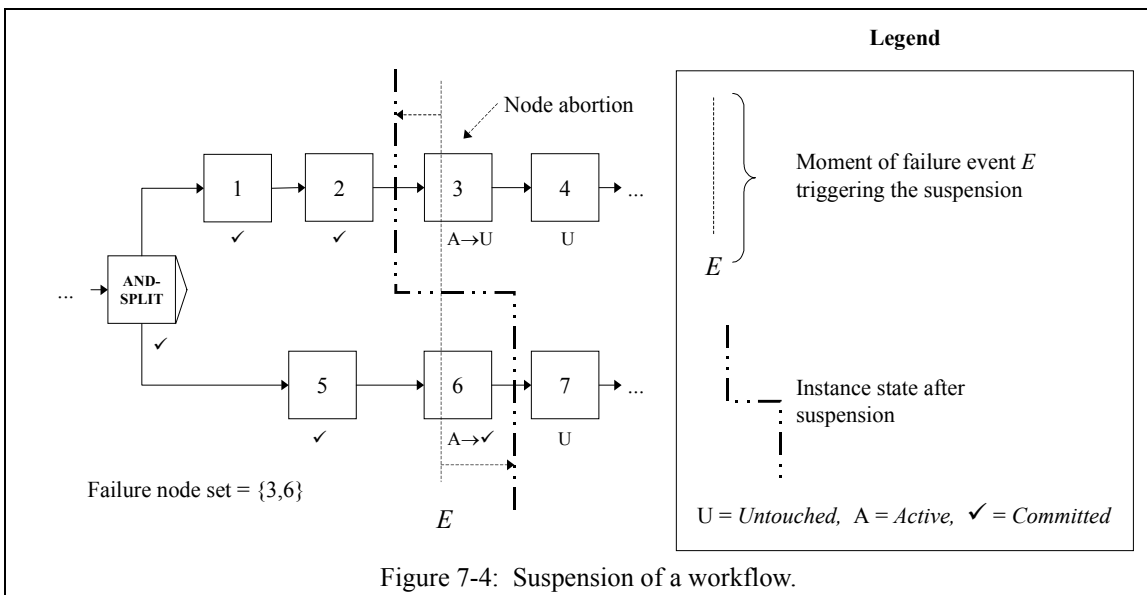


Figure 7-4: Suspension of a workflow.

ately. Therefore, the path node 3 belongs to is rolled back to the already committed node 2. In particular, node 3 is set from state *Active* to state *Untouched* (via state *Control-Flow-Failed*; see 5.4.2⁴). In contrast to this, node 6 may execute an uncritical activity (e.g., a diagnostical examination) and therefore may be allowed to commit (node transition *Active* \rightarrow *Committed*) before its path is suspended.

In particular, this means that a workflow instance *I* may temporarily be in the two states *Active* and *Rolling-Back* simultaneously. After the rollback respective the commit of the currently executed nodes, *I* is set to state *Suspended*. Additionally, the user has to specify whether the time that is needed to commit or abort currently active nodes has to be subtracted from the time the instance has to be suspended, or not.

7.4.3 Processing of Local Control Actions

The principal way of processing local control actions has already been described in Chapter 3. In particular, the criteria for selection reactive or predictive selection have been described there. Thus, we can now concentrate on more specific details: For *non-additive* local control actions, we can now define formally on the base of the workflow execution model of Chapter 5 when a node is *affected by a control action* and when not (7.4.3.1). As an important subproblem, we furthermore have to specify what has to be done if a currently executed node (i.e., a node in state *Active* of the failure node set) is affected by such a non-additive control action (7.4.3.2). For local *additive* control actions, we have to describe how AGENTWORK decides into which workflow an additional node should be inserted (7.4.3.3).

Note that we do *not* yet describe the *structural* adaptation of a workflow on the node and edge level, such as dropping or adding nodes. This is a matter of Chapter 8 (*Structural Adaptation Operators*). Rather, in this chapter we describe “only” the step between control action triggering and the final structural workflow adaptation. In particular, in this section we concentrate only on a *single* control action. Possible dependencies between control actions that have overlapping valid time intervals are discussed later in Section 7.6.

7.4.3.1 Processing of Non-Additive Local Control Actions

Non-additive local control actions affect activity nodes which are already contained in a workflow definition. To describe their processing, we introduce the following notations:

- $ca(A, C)$ resp. $ca(R)$ denotes an arbitrary non-additive case-related resp. resource-related control action from Table 7-2. Parameter *A* denotes the activity pattern of the control action, parameter *C* the case instance, and *R* the resource pattern. Additional parameters – such as the *d*-parameter of $postpone(A, d, C)$ – are irrelevant for this section, so that we can omit them here.
- *VT* denotes the fixed or conditional valid time of $ca(A, C)$ resp. $ca(R)$.

4. Recall from this section that the node state *Control-Flow-Failed* expresses an activity node “abortion” due to some control flow failure.

- If n is an activity node, then n^i denotes the i -th execution of n ($i = 1, 2, \dots$) during the execution of the workflow instance to which n belongs. Parameter i may be larger than 1 if n is located in a loop, or when n is executed twice due to some partial rollback.

Then, (*) states under which conditions an activity node execution is affected by a control action.

Affected Activity Node Execution

(*)

An activity node execution n^i is affected by a non-additive local control action $ca(A, C)$ resp. $ca(R)$ if the following conditions hold:

1. n^i is started during VT . This means that for the i -th execution of n it holds that n is set to state *Active* during VT , i.e., that $entry-of-node-state(n^i, Active) \in VT$

If the control action is *case*-related, it holds:

2. n is an *A*-node. Formally, this means that the pattern defined in the *activity* entry of $NAM(n)$ matches A .
3. The *Case* instance C is identical with the *Case* instance assigned to n^i . Formally, this means that it holds: $C = case(n^i)$

If the control action is *resource*-related, it holds:

- 2'. At least one resource instance assigned to n^i matches pattern R . Formally, it holds that it exists $r \in resources(n^i)$ with r matching R .

Based on this, we can now precisely formulate the selection criteria for predictive and reactive adaptation as follows:

Predictive adaptation is selected for a node execution n^i under the following conditions:

- VT is fixed (condition 1 of 3.4.1).
- For a node execution n^i it can be determined at the point in time when $ca(A, C)$ resp. $ca(R)$ has been triggered, that n^i is affected by $ca(A, C)$ resp. $ca(R)$ according to (*). This is the precise formulation of condition 2 in 3.4.1.

In particular, this requires that the workflow can be estimated (starting at the failure node set) to determine whether the execution n^i will take place during VT . If this estimation is possible and the conditions listed in (*) hold, control flow operators can be invoked predictively to adapt the control flow, e.g., to drop or postpone the execution n^i (by dropping or postponing node n in the control flow).

Reactive adaptation is selected when it cannot be determined predictively whether a node execution n^i meets the conditions listed in (*) or not. Then, two principal mechanisms exist *when* to

determine reactively whether n^i meets the conditions of (*) or not:

- *On-demand check*

This straightforward mechanism checks the conditions of (*) when n is set to state *Active* (w.r.t. its i -th execution). This makes sense, as at this moment the case and the resources already have been assigned to n (see 5.4.3.5), and as at this moment it is definitely known by definition whether n has been set to state *Active* during *VT* or not. If the conditions of (*) are met, the execution of n is directly terminated and then handled according to the affecting control action (e.g., n is dropped in case of a *drop* control action). However, this mechanism has two disadvantages: First, activity executions that already have been presented in the user worklist and for which the user has confirmed that they shall be started, suddenly may be declared as failed activities (as they are affected by a control action). This may confuse workflow users. Second, as the node execution is checked *after* n has been set to state *Data-Activated*, the input data already have been provided at this moment. If this data have been obtained by user input and if for instance n is affected by a *drop* or *replace* control action, this would mean that the user has typed in the data for nothing.

- *Advanced check*

This mechanism already reacts when n is set to state *Control-Activated* (w.r.t. its i -th execution). At this moment, at least the case- resp. resource-related conditions of (*) can be checked. Concerning the question whether n will be to state *Active* during *VT* or not, this mechanism assumes that

$$\text{entry-of-node-state}(n^i, \text{Control-Activated}) \in VT \Rightarrow \text{entry-of-node-state}(n^i, \text{Active}) \in VT \quad (xii)$$

This assumption makes sense as valid time intervals typically cover days or even weeks, while the durations of node state transitions on the average are much smaller, i.e., only take minutes or hours. This mechanism avoids the disadvantages of the on-demand check, in particular use-less data input can be avoided. However, it has the disadvantage that assumption (xii) may not hold for some particular node execution. Thus, workflow monitoring is required when the workflow is continued.

The question which of the two mechanisms is the better one depends on the particular workflow application. For example, if a workflow application is characterized by interactive data input for many activity types, one may prefer the advanced check. If it cannot be assumed that assumption (xii) holds very often, one may prefer on-demand check. Therefore, the used mechanism has to be specified when AGENTWORK is installed for a particular workflow application.

Independently from the particular check mechanism, the same control flow operators as for predictive adaptation are invoked for example to drop or postpone the execution n^i (see Chapter 8).

The way AGENTWORK selects its adaptation strategy also shows that predictive or reactive adaptation is a matter of single activity node execution, and not of the execution of whole workflow parts. In other words, for a workflow part some nodes may be handled predictively, as the necessary

conditions can be checked at the point in time of control action triggering, while other nodes of the same workflow part have to be handled reactively.

7.4.3.2 Processing of Currently Executed Activity Nodes

An open problem is how to handle an activity node n which already is in state *Active* at the point in time a non-additive control action has been triggered, *and* which is affected by this control action according to (*). Analogously to workflow suspension, it is difficult to find a general mechanism how to deal with such active nodes. For example, imagine that $replace(A,B,C)$ has been triggered and that A and B are some drug administration patterns including a dosage specification. If a currently executed A -node has been started only several minutes before control action triggering, there may be no problem to directly abort the A -node, and to administer the new drug with the dosage specified by B . However, if the execution of the A -node already has started some time ago so that already a significant part of the dosage has been administered to the patient identified by C , it may be more appropriate to give only a part of the dosage specified by B to avoid toxicity effects for the patient. As it cannot be automatically decided which particular mechanism is more appropriate, AGENTWORK therefore requests an authorized user whenever an active node is affected by a non-additive control action. This user then has to state whether the control action affecting n shall be realized as specified in the triggering failure rule, or whether it for instance shall be modified for this particular node. For example, for the $replace(A,B,C)$ scenario described above the physician could reduce the dosage specified by B significantly to avoid toxicity effects for the patient.

7.4.3.3 Processing of Additive Local Control Actions

The additive control actions $add(A,C)$ and $add-repetitively(A,d,C)$ state that A -nodes have to be executed *additionally*. In this section, we discuss how to identify an appropriate workflow W into which a new A -node for case C can be inserted. The question *where* within W a new A -node shall be inserted depends on the structure of W in the neighborhood of the failure node set, and is a matter of operators which adapt the control flow. This will be described in Chapter 8 (*Structural Adaptation Operators*).

Concerning the selection of an appropriate workflow, it principally should be avoided to insert activity nodes into workflows executing “unrelated” activities. For example, in the medical domain a radiological workflow usually is not an appropriate candidate to execute additional drug administrations as then one workflow would involve more than one medical department or would at least confuse the medical staff. To determine an appropriate workflow for a new A -node, AGENTWORK provides two principal mechanisms, namely manual selection with case-based ordering and workflow scope labeling.

Manual selection with case-based ordering

This straightforward mechanism generates a user request when an $add(A,C)$ or $add-repetitively(A,d,C)$ control action has been triggered. All currently executed workflows are presented to an authorized user who has to decide to which workflow a new A -node shall be added. The way the

list of currently executed workflows is presented to the user is as follows:

The first sublist of the list presented consists of all workflows executed *exclusively* for case C . This is because it is viewed as reasonable to insert a new A -node for case C into a workflow executing its other activities for C as well, and not for another case C' . Thus, AGENTWORK presents the workflows of this sublist first to the user.

The second sublist of the list consists of all workflows for which it is known – at the point in time when $add(A, C)$ or $add-repetitively(A, d, C)$ has been triggered – that *some* of its activity nodes are executed for case C . The user can insert a new A -node into a workflow of this second sublist, if there are no workflows running exclusively for case C , or when the user for some reason views none of the workflow running exclusively for case C as appropriate.

The last sublist of the list consists of the remaining currently executed workflows. The user inserts a new A -node into a workflow of this last sublist, if sublists 1 and 2 above are empty, or when the user for some reason views none of the workflow of sublists 1 or 2 as appropriate.

Workflow scope labeling

To reduce the number of candidates during workflow selection, AGENTWORK also allows to label workflow definitions with an “activity scope”. The labels that are used for this are directly derived from the *Activity* subclasses of the global data schema (see 4.2.1) as follows: For every *Activity* subclass X , the label X -Workflow is provided (e.g., for the class *Therapeutic-Activity* the label *Therapeutic-Activity-Workflow* is provided). At workflow definition time, the user then can assign such an X -Workflow label to a workflow definition if “most” of the used activity definitions are derived from class X , or, more formally, if it holds for “most” activity definitions assigned to activity nodes that the *activity* entry is of type *Obj-Patt*< X >; see 5.3.1). The question what “most” means (e.g., 80 or 90 percent) is left up to the user. In particular, it is not required that the *activity* entry of *all* activity definitions of the workflow definition is of type *Obj-Patt*< X >. This is because as it still makes sense to label a workflow definition which uses for example 12 different activity definitions as *Therapeutic-Activity-Workflow*, even if only 9 or 10 activity definitions have an *activity* entry of type *Obj-Patt*<*Therapeutic-Activity*>.

Then, when an A -node has to be added at workflow execution time because of an $add(A, C)$ or $add-repetitively(A, d, C)$ control action, AGENTWORK suggests only those currently executed workflows which have a label X -Workflow for which A matches X , or, precisely, for which it holds:

$$Y \quad IS-A \quad X, \quad \text{if } A \text{ is of type } Obj-Patt<Y>. \quad (xiii)$$

This means that the activity pattern A of the new node fits into the workflow activity scope described by X .

The list of workflows fulfilling constraint (xiii) is then presented to the user for further selection, and is ordered the same way as it is done for manual selection with case-based ordering. If the list of workflows fulfilling constraint (xiii) is empty, AGENTWORK shifts to the manual selection with case-based ordering mechanism.

If there is no currently executed workflow at all, or if no currently executed workflow for some reason is viewed as appropriate for executing an additional A -node, AGENTWORK generates a new workflow instance which contains only a START and END node. Into this “empty” workflow the control flow operators of Chapter 8 can insert the required new A -nodes. However, this mechanism is viewed only as an “emergency solution” as such artificial workflows with only one activity type should be avoided.

Concerning the selection of predictive or reactive adaptation, the following criteria are used for additive control actions:

For $add(A, C)$, predictive adaptation is used by default. Recall from Section 7.2.2, that the end of a valid time interval VT assigned to an $add(A, C)$ control action cannot be specified by a termination condition, so that the “knock-out” criterion for predictive adaptation does not exist for $add(A, C)$. Predictive adaptation for $add(A, C)$ means, that the workflow part corresponding to the valid time VT assigned to $add(A, C)$ is estimated, and that the control flow adaptation operators insert the new A -node into this part. If such an estimation is not possible for some reason, reactive adaptation is selected. This means that the new A -node is inserted as “close as possible” to a member of the failure node set. By this, it is achieved that the execution of a new A -node can be started as soon as possible after the adapted workflow path is continued, so that the constraint that the execution of a new A -node has to be started during VT can be met. The conflict situation that the execution of a new A -node cannot be started anymore during VT due to some unexpected delays is handled by workflow monitoring.

The control action $add-repetitively(A, d, C)$ is always handled by a specific subtype of predictive adaptation, even if the valid time VT assigned to $add-repetitively(A, d, C)$ is terminated conditionally. Predictive adaptation for $add-repetitively(A, d, C)$ is realized by inserting a *loop* of A -nodes with period d into the selected workflow. The termination condition of the loop is exactly the expiration of the valid time VT , i.e., the loop terminates when VT terminates. This is a form of predictive adaptation as all execution of A -activities specified by $add-repetitively(A, d, C)$ are inserted predictively at the point in time when $add-repetitively(A, d, C)$ has been triggered. For the generation and insertion of such a loop on the node and edge level we again refer to Chapter 8.

The alternative to generating a loop would be to insert several sequential A -nodes into the workflow is not viewed as appropriate. For example, imagine that

$add-repetitively(A, (1, \text{week}), C) \text{ VALID-TIME } [now, now + (6, \text{week})]$

has been triggered to express that an A -node has to be executed weekly during the next 6 weeks. Without using loops, one would have to estimate which part of the remaining workflow corresponds to week 1, which to week 2 and so on, and to insert a new A -node into each of these parts. It is unlikely that such an estimation will have a sufficient precision so that AGENTWORK does not make use of this possibility.

7.5 Integrity of Failure Rules

The problem of rule integrity is a topic of ongoing research in the fields of databases and artificial intelligence [BARALIS 1999, PERRAJU & PRASAD 2000, ROANES-LOZANO ET AL. 2000], and cannot be addressed in general in this thesis. However, as we have introduced some significant restrictions and simplification conventions concerning failure rule definitions, we can handle at least the most important types of rule integrity for AGENTWORK. We recall from 4.4.2 that the main rule restriction is that the *THEN* part (without the valid time) contains only *one* control action⁵, and no additional elements such as data manipulation statements or procedure calls. Furthermore, to facilitate failure handling we have introduced two conventions for the valid time interval of a control action, namely the beginning at *now* and *coherence* (Section 7.3). In the following, we discuss several aspects of rule integrity, in particular rule redundancy (7.5.1), rule incompatibility (7.5.2), and rule termination (7.5.3).

7.5.1 Rule Redundancy

As an important prerequisite for rule integrity, it is desirable to avoid redundant rules to reduce the number of rules and thus to facilitate the maintenance, analysis and processing of rules. For example, this is important for a domain such as hemato-oncology where several hundred failure rules exist in the medical text books for treatment adaptation. Due to the bipartite event-action structure of ACTIVETFL failure rules, we distinguish two basic redundancy types, namely *event redundancy* and *action redundancy*. In this section we discuss only *static* redundancy, i.e., redundancy that can be already identified at rule definition time. *Dynamic* redundancy that occurs at rule execution time is discussed later in Section 7.6 (*Dynamic Control Action Dependencies*).

7.5.1.1 Event Redundancy

Let us assume two rules R_I and R_2 with identical *THEN* part (i.e., the triggered control actions, their parameters and their valid times are identical). Then R_2 is said to be *event-redundant* w.r.t. R_I , if any event that triggers R_2 also triggers R_I . An example is

$$\begin{array}{lll}
R_j: & \text{WHEN} & \text{INSERT ON hemato-findings} \\
& \text{WITH} & \text{new.parameter} = \text{Leukocyte-Count AND new.value} < 1000 \\
& \text{THEN} & \text{drop}(A, C_{\text{new}})
\end{array} \tag{xiv}$$

R_2 : WHEN INSERT ON hemato-findings
 WITH new.parameter = Leukocyte-Count AND new.value < 500
 THEN drop(A, C_{new}). (xv)

5. A control action may also appear in the valid time, e.g., *THEN add-repetitively(A,d,C) VALID-TIME [now, drop(D,C)]*. However, this is irrelevant for our considerations concerning rule integrity.

R_2 is event-redundant w.r.t. R_1 , as any leukocyte count triggering R_2 also triggers R_1 . Thus, R_2 can be dropped from the rule base. The general mechanism to identify such event redundant rules is as follows:

- First, identify all rule pairs R_1, R_2 with identical *THEN* part.
- Second, check whether the event triggering R_1 subsumes the event triggering R_2 . This is given, if both *WHEN* parts refer to the same extension and extension operation (e.g., *hemato-findings* and *INSERT ON* in R_1 and R_2), and if the pattern in the *WITH* part of R_1 subsumes the pattern in the *WITH* part of R_2 according to Section 4.2.1.6 (*Pattern Subsumption and Pattern Matching*).

7.5.1.2 Action Redundancy

We first describe action redundancy for *non-additive* local control actions, and second for *additive* local control actions. Action redundancy for global control actions cannot occur as an *abort(W,C)* cannot imply a *suspend(W,C)* and vice versa. The same holds for the pair *abort-workflows-of(R)* and *suspend-workflows-of(R)*.

For all rule pairs R_1 and R_2 used in this section we assume identical *WHEN* and *WITH* parts.

Action redundancy for non-additive local control actions:

We first consider case-related non-additive control actions. For this, let

$$ca_1(A_1, [B_1, d_1, p_1, f_1] C_1) \text{ VALID-TIME } VT_1 \quad \text{and} \\ ca_2(A_2, [B_2, d_2, p_2, f_2] C_2) \text{ VALID-TIME } VT_2$$

denote the case-related non-additive control actions of the *THEN* part of two rules R_1 and R_2 (the parameters B_i, d_i, p_i and f_i are only needed for $ca_i = \text{replace, postpone or change-value}$). Then R_2 is said to be *action-redundant* w.r.t. R_1 , if the following conditions hold:

1. $ca_1 = ca_2, C_1 = C_2$, and if
 - $ca_i = \text{replace}: B_1 = B_2$
 - $ca_i = \text{postpone}: d_1 = d_2$
 - $ca_i = \text{change-value}: p_1 = p_2 \text{ and } f_1 = f_2$
2. A_2 matches A_1 , or – equivalent to this – A_1 subsumes A_2 (according to 4.2.1.6), and VT_2 is contained in VT_1 .

If conditions 1 and 2 hold, we also say that the control action ca_2 is *redundant* w.r.t. ca_1 . An example is the following: If we have

$$A_1 := \text{Drug-Administration}[drug = \text{"ETOPOSID"}] \\ A_2 := \text{Drug-Administration}[drug = \text{"ETOPOSID"}, dosage > 150, unit = mg]$$

and the two rules (by omitting the identical *WITH* parts)

$$R_1: \quad \text{WHEN } E \quad (xvi) \\ \text{THEN } \text{drop}(A_1, P) \text{ VALID-TIME } [now, now + (3, day)]$$

$$R_2: \quad \text{WHEN } E \quad (xvii) \\ \text{THEN } \text{drop}(A_2, P) \text{ VALID-TIME } [now, now + (2, day)],$$

then rule R_2 is action-redundant w.r.t R_1 , as any dropping of an ETOPOSID administration during the next three days implicitly means also a dropping of any ETOPOSID administration with more than 150 mg during the next two days. Thus, R_2 can be dropped from the rule base.

A special type of redundancy for case-related non-additive control actions is given if we do not have $ca_1 = ca_2$ in condition 1, but $ca_1 = \text{replace}$ and $ca_2 = \text{drop}$. Then a $\text{replace}(A_1, X, C_1)$ control action would make a $\text{drop}(A_2, C_2)$ control action superfluous, as replacing A_1 -activities in particular implies that A_2 -activities are not executed anymore (given that A_2 matches A_1 due to condition 2), i.e., that they are dropped.

For resource-related non-additive control actions, redundancy is defined analogously. For this, let

$$ca_1(R_1, [d_1]) \text{ VALID-TIME } VT_1 \quad \text{and} \\ ca_2(R_2, [d_2]) \text{ VALID-TIME } VT_2$$

denote the resource-related non-additive control actions of the *THEN* part of two rules R_1 and R_2 (the parameter d_i is only needed for $ca_i = \text{postpone-activities-of}$). Then R_2 is said to be *action-redundant* w.r.t. R_1 , if the following conditions hold:

1. $ca_1 = ca_2$, and if
 $ca_i = \text{postpone-activities}: d_1 = d_2$
2. R_2 matches R_1 , or – equivalent to this – R_1 subsumes R_2 (according to 4.2.1.6), and VT_2 is contained in VT_1 .

Action redundancy for additive local control actions: Let

$$ca_1(A_1, [d_1] C_1) \text{ VALID-TIME } VT_1 \quad \text{and} \\ ca_2(A_2, [d_2] C_2) \text{ VALID-TIME } VT_2$$

denote the additive control actions of the *THEN* part of two rules R_1 and R_2 (the parameter d_i is only needed for $ca_i = \text{add-repetitively}$). Then R_2 is said to be *action-redundant* w.r.t. R_1 , if the following conditions hold:

1. $ca_1 = ca_2$, $C_1 = C_2$, and if
 $ca_i = \text{add-repetitively}: d_1 = d_2$

2. A_1 matches A_2 , or – equivalent to this – A_2 subsumes A_1 (according to 4.2.1.6), and VT_2 is contained in VT_1 .

Note that concerning the matching relationship of the used activity patterns, condition 2 is just the *opposite* direction if compared with condition 2 for non-additive case-related control actions. This is because *adding* the more specific activity (i.e., A_1 in condition 2) also implies that the more general activity (i.e., A_2 in condition 2) is added. For example, if we have

$$A_1 := \text{Drug-Administration}[\text{drug} = \text{"ETOPOSID"}, \text{dosage} > 150, \text{unit} = \text{mg}]$$

$$A_2 := \text{Drug-Administration}[\text{drug} = \text{"ETOPOSID"}, \text{dosage} > 100, \text{unit} = \text{mg}]$$

(i.e., A_2 subsumes A_1) and the two rules

$$\begin{array}{ll} R_1: & \text{WHEN } E \\ & \text{THEN } \text{add}(A_1, P) \quad \text{VALID-TIME } [\text{now}, \text{now} + (3, \text{day})] \end{array} \quad (\text{xviii})$$

$$\begin{array}{ll} R_2: & \text{WHEN } E \\ & \text{THEN } \text{add}(A_2, P) \quad \text{VALID-TIME } [\text{now}, \text{now} + (2, \text{day})], \end{array} \quad (\text{xix})$$

then rule R_2 is action-redundant w.r.t R_1 as adding an ETOPOSID administration with more than 150 mg for the next three days (R_1) implicitly means that an ETOPOSID administration with more than 100 mg is added for the next two days (R_2). Thus, rule R_2 can be dropped from the rule base.

A special type of redundancy for case-related additive control actions is given if we do not have $ca_1 = ca_2$ in condition 1, but $ca_1 = \text{add-repetitively}$ and $ca_2 = \text{add}$, for example if we use $\text{add-repetitively}(A_1, d_1, P)$ in (xviii) instead of $\text{add}(A_1, P)$. Then, an $\text{add-repetitively}(A_1, d_1, C_1)$ control action during a valid time VT_1 implicitly means that (at least) one A_2 -activity is added during VT_2 (as VT_1 contains VT_2) so that the rule with the $\text{add}(A_2, d_2, C_2)$ control action is superfluous.

As the definitions for event and action redundancy show, the redundancy of a rule R_2 w.r.t R_1 is a *non-symmetric* relationship, i.e., if R_2 is redundant w.r.t R_1 and thus can be dropped from the rule base, R_1 of course is not redundant w.r.t R_2 .

7.5.1.3 Other Event/Action Redundancy Constellations

In Section 7.5.1.1 and Section 7.5.1.2, we assumed that either the *THEN* parts of two rules are identical (for considering event redundancy), or that the *WHEN* / *WITH* parts of two rules are identical (for considering action redundancy). Of course, there may be redundancy effects for two rules also if these assumptions do not hold. The most obvious constellation in this context is the following: Let us assume two following two rules with non-additive control actions:

$$\begin{array}{ll} R_1: & \text{WHEN } E_1 \\ & \text{THEN } ca_1 \end{array} \quad (\text{xx})$$

$$\begin{array}{ll}
 R_2: & \text{WHEN } E_2 \\
 & \text{THEN } ca_2
 \end{array}
 \tag{xxi}$$

where E_1 subsumes E_2 , and where the control actions ca_1 and ca_2 meet the conditions listed for action redundancy w.r.t. non-additive local control actions in Section 7.5.1.2. Then, rule R_1 is event- *and* action-redundant w.r.t R_2 and therefore can be dropped from the rule base. For additive local control actions, simultaneous event- *and* action-redundancy is defined analogously.

Other constellations concerning the subsumption of events and actions are difficult to interpret in terms of redundancy, so that they are not handled automatically by AGENTWORK. In particular, if the end of some valid times VT_1 or VT_2 in two rules is defined *conditionally* via *Until* or *Unless*, it is difficult at rule definition time to determine whether VT_1 covers VT_2 or vice versa. Furthermore, *partial* redundancy is also difficult to handle. For example, let us assume that the valid time of R_1 at (xvi) would be 2 days (instead of 3 days), and that of R_1 at (xvii) 3 days (instead of 2 days). This would state that during the first two days *all* ETOPOSID administrations shall be dropped, and during the third day only those with a dosage higher than 150 mg. Thus, R_2 would be *action-redundant* w.r.t. R_1 for the first two days of the valid time, but not for the third day. However, such a partial redundancy may be intended by the rule modeler. As such partial redundancy may become arbitrarily complex and because rule consistency is not the focus of this thesis, we do not address this topic here.

7.5.2 Rule Incompatibility

In general, the term *rule incompatibility* refers to the situation that two rules trigger incompatible statements at the same point in time. In our specific context, this means that two failure rules trigger incompatible control actions with overlapping valid time intervals. For example, if two rules trigger a $drop(A, C)$ and an $add(A, C)$ control action with the same activity pattern A for the same case C with overlapping valid time intervals for the same event, this does not make sense and thus has to be avoided. In this section we concentrate on *static incompatibility* which can be detected at rule definition time. *Dynamic* incompatibility which occurs at rule execution time is discussed in Section 7.6 (*Dynamic Control Action Dependencies*).

We first define rule incompatibility for global control actions. Second, we define rule incompatibility for rule pairs where both rules consist of case-related local control actions. Third, we define rule incompatibility for rule pairs where both rules consist of resource-related local control actions. Whenever such an incompatibility is detected at rule definition time, the rule modeler manually has to resolve the incompatibility by dropping or editing the respective rules.

Incompatibilities between rules where one rule consists of a case-related local control action and the other of a resource-related control action can only be identified at rule execution time and thus are discussed in Section 7.6.

In the following, we assume for all considered rule pairs R_1, R_2 that they have overlapping event patterns, i.e., that an object insert or update may trigger both rules simultaneously.

7.5.2.1 Incompatibility between Global Control Actions

Two rules with global control actions and overlapping event patterns are statically incompatible, if one rule triggers an *abort*(W, C), and the other a *suspend*(W, C) referring to the same workflow definition and the same case. This is because it does not make much sense that the same object insertion or update may trigger the abortion *and* the suspension of the same workflow instance. Furthermore, two rules are also statically incompatible if one rule triggers an *abort-workflows-of*(R), and the other a *suspend-workflows-of*(R) referring to the same resource pattern.

7.5.2.2 Incompatibility between Case-Related Control Actions

Let

$$ca_1(A_1, [B_1, d_1, p_1, f_1] C_1) \text{ VALID-TIME } VT_1 \quad \text{and} \\ ca_2(A_2, [B_2, d_2, p_2, f_2] C_2) \text{ VALID-TIME } VT_2$$

denote the case-related control actions of the *THEN* part of two rules R_1 and R_2 with overlapping event patterns (the parameters B_i, d_i, p_i and f_i are only needed for $ca_i = \text{replace, postpone, change-value or add-repetitively}$). Then R_2 is said to be *statically incompatible* with R_1 (w.r.t. $VT_1 \cap VT_2$), if the following conditions hold:

1. C_1 and C_2 refer to the same case, and the valid time intervals VT_1 and VT_2 overlap .
2. A_1 subsumes A_2
3. The control actions ca_1 and ca_2 are incompatible according to Table 7-4.

For selected control action pairs in Table 7-4, we explain why they are viewed as incompatible or compatible (for the other pairs analogous arguments hold). In the following, for a pair (i, j) , i refers to the row, and j to the column of Table 7-4. Furthermore, to avoid an overlapping of the different cells of Table 7-4, we agree on the convention that for any cell *above* the grey diagonal the *equality* of the patterns A_1 and A_2 is allowed, but forbidden for any other cell (i.e., the equality case for A_1 and A_2 is covered by the cells above the grey diagonal). Note further, that the matrix of Table 7-4 is not symmetrical due to the subsumption relationship between A_1 and A_2 (i.e., in general A_1 and A_2 are *not* identical patterns).

Pair (1,3) *drop*(A_1), *postpone*(A_2, d_2): This pair is viewed as incompatible, as on one side it is specified that A_2 -activities shall be postponed, but on the other side shall be dropped due to *drop*(A_1) (as A_1 subsumes A_2).

Pair (3,1) *postpone*(A_1, d_1), *drop*(A_2): This pair is viewed as compatible as on one side it is specified that A_1 -activities shall be postponed, but that only *some* of them (i.e., the A_2 -activities) shall be dropped. For example, let us assume

		1	2	3	4	5	6
	ca_2	$drop(A_2)$	$replace(A_2, B)$	$postpone(A_2, d_2)$	$change-value(A_2, p_2, f_2)$	$add(A_2)$	$add-repetitively(A_2, d_2)$
ca_1							
1	$drop(A_1)$	CP	CP* (ICP if $A_1 \supset B^{**}$)	ICP	ICP	ICP	ICP
2	$replace(A_1, B)$	CP	CP	ICP	ICP	ICP	ICP
3	$postpone(A_1, d_1)$	CP	CP	CP (ICP only for $d_1 \neq d_2$)	CP	CP***	CP***
4	$change-value(A_1, p_1, f_1)$	CP	CP	CP	CP	CP	CP
5	$add(A_1)$	CP	CP	CP	CP	CP	CP
6	$add-repetitively(A_1, d_1)$	CP	CP	CP	CP	CP	CP (ICP only for $d_1 \neq d_2$)
ICP = Incompatible, CP = Compatible.							
<p>* It is <i>not</i> viewed as incompatible that a subset of the A_1-activities to be dropped (namely the A_2-activities) shall be replaced by B-activities.</p> <p>** As the new B-activities would directly have to be dropped due to $drop(A_1)$.</p> <p>*** Order to be determined (manually) at execution time.</p>							

Table 7-4: Incompatibility table for case-related control actions (with A_1 subsuming A_2).
The C_i -parameters have been omitted as due to condition 1 all control actions refer to the same case (e.g., the same patient). The *review* control action is not listed as it has to be manually transformed to one of the other case-related control actions.

$A_1 := Drug-Administration[drug = "ETOPOSID"]$

$A_2 := Drug-Administration[drug = "ETOPOSID", dosage > 150, unit = mg].$

Then, $postpone(A_1, d_1)$ and $drop(A_2)$ mean that the ETOPOSID administrations principally have to be postponed but that some of them (i.e., those with a dosage higher than 150 mg) have to be dropped. From the medical point of view, such a combination makes sense and thus should not generally be viewed as incompatibility. Therefore, AGENTWORK searches for such combinations and informs the rule modeler about them, but does not forces the rule modeler to re-edit such rules.

Pair (3,5) $postpone(A_1, d_1)$, $add(A_2)$: This pair is viewed as compatible, as an A_2 -activity can be added to A_1 -activities that shall be postponed. However, if this pair is triggered the *order* in which both control actions shall be processed has to be determined at execution time, i.e., whether the A_2 -activity shall be added first and then postponed with all A_1 -activities, or whether the A_2 -activity shall be added after the A_1 -activities have been postponed.

Pair (1,5) $drop(A_1)$, $add(A_2)$: This pair is viewed as incompatible, as on one side it is specified that an A_2 -activity shall be added, but on the other side also shall be dropped due to $drop(A_1)$ (as A_1 subsumes A_2).

Pair (5,1) $add(A_1)$, $drop(A_2)$: This pair is viewed as compatible as adding a more general activity (i.e., an A_1 -activity) does not exclude that more specific activities are dropped from a workflow. For example, if we have

$$A_1 := \text{Radiodiagnostic-Activity}[focus = \text{“Liver”}] \quad \text{and} \quad (xxii)$$

$$A_2 := \text{MRT-Examination}[focus = \text{“Liver”}]^6 \quad (xxiii)$$

this would mean that a radiodiagnostic activity shall be added (e.g., a computer tomography examination), but that any mrt examination shall be dropped. The only incompatibility that may occur is that at control action triggering time A_1 is changed or refined manually (see Section 7.2.3.4), e.g., becomes

$$A_1 := \text{MRT-Examination}[focus = \text{“Liver”}]. \quad (xxiv)$$

This is also detected by AGENTWORK, as the equality of A_1 and A_2 caused by the refinement of A_1 as shown in (xxiv) is not covered anymore by pair (5,1), but by pair (1,5) (as the equality of patterns A_1 and A_2 in Table 7-4 is only allowed in the cells above the grey diagonal).

7.5.2.3 Incompatibility between Resource-Related Control Actions

Two rules with overlapping event patterns and resource-related control actions are viewed as incompatible, if one rules triggers a *drop-activities-of(R)* control action, and the other one a *post-pone-activities-of(R,d)* control action for the same resource.

7.5.3 Rule Termination

Generally, for a rule base it should be guaranteed that for any data constellation rule processing cannot continue forever, i.e., that rules cannot activate each other indefinitely. Though our failure rules have a very restricted structure (e.g., only one control action in the *THEN* part), the problem of rule termination also has to be considered for our rule base of failure rules. This is because control actions may also appear in the *WHEN* part of a rule, as they formally are predicates and thus F-Logic formulas (see Section 4.2.3 and 4.2.5). Thus, cycles principally can occur. For example, if we have

$$A_1 := \text{Drug-Administration}[drug = \text{“ETOPOSID”}]$$

$$A_2 := \text{Drug-Administration}[drug = \text{“DOXYCYCLIN”}]$$

there may be the rule

6. Recall from Section 4.2.1 that *MRT-Examination* is a subclass of *Radiodiagnostic-Activity*.

$$\begin{array}{ll} R: & \text{WHEN } \text{drop}(A_1, C) \text{ VALID-TIME } [now, now + (n, day)] \\ & \text{THEN } \text{drop}(A_2, C) \text{ VALID-TIME } [now, now + (n, day)] \end{array} \quad (xxv)$$

stating that whenever the drug ETOPOSID is dropped for n days, the drug DOXYCYCLIN can also be dropped for the same time. The rationale for this is that often an antibiotic drug is given in parallel to immunosuppressive drugs such as ETOPOSID to prevent bacterial infections. Thus, when the immunosuppressive drug is dropped the antibiotic drug can also be dropped. If a rule triggered by (xxv) then would trigger a $drop(A_i, C)$, this would lead to a cycle which should be avoided.

As the control flow failures introduced in this thesis do not add any additional complexity to rule processing as known from active databases and temporal logics, and as this thesis does not focus on rule processing, we refer to the literature for rule termination analysis. In particular, [BARALIS 1999] provides a comprehensive overview on termination analysis in active databases, and [GABBAY ET AL. 1998] for the same topic for logical languages.

7.6 Dynamic Control Action Dependencies

We now discuss *dynamic* dependencies between control actions, i.e., dependencies between control actions that can only be determined at rule execution time. We first discuss dependencies between global control actions (7.6.1), second between global and local control actions (7.6.2), and third between local control actions (7.6.3). In particular, for local control actions dynamic redundancy and dynamic incompatibility effects are discussed.

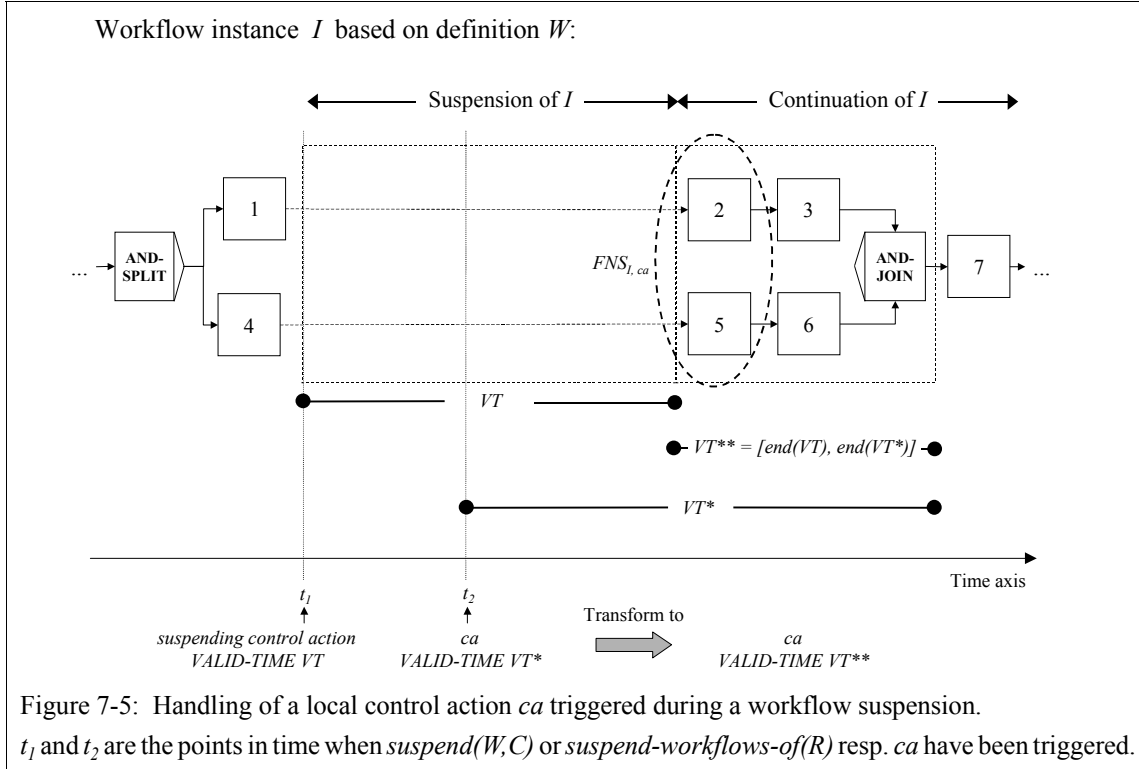
7.6.1 *Dynamic Dependencies between Global Control Actions*

The only situation that has to be considered is that an *abort(W,C)* or *abort-workflows-of(R)* control action is triggered during the valid time of a *suspend(W,C)* or *suspend-workflows-of(R)* control action, and that both triggered control actions affect the same workflow instance. For example, if an *abort-workflows-of(R)* is triggered during the valid time of a *suspend(W,C)* control action, then a workflow instance that needs a *Resource* instance matching pattern *R* is affected by *abort-workflows-of(R)*. Coincidentally, this instance may also be based on the workflow definition *W* and executed for case *C*, so that it is also affected by *suspend(W,C)*.

The handling of such a situation is straightforward: AGENTWORK assumes that a control action stating the abortion of a workflow always has a higher priority than a control action stating the suspension of a workflow. This is because AGENTWORK assumes that an *abort(W,C)* or *abort-workflows-of(R)* is only triggered when events have occurred that make the continuation of a workflow impossible. Thus, whenever an *abort(W,C)* or *abort-workflows-of(R)* control action is triggered during the valid time of a *suspend(W,C)* or *suspend-workflows-of(R)* control action, and both triggered control actions affect the same workflow instance, this workflow instance is aborted.

7.6.2 *Dynamic Dependencies between Global and Local Control Actions*

As a single workflow may be affected both by a global control action *and* a local control action,



several dependencies between both may occur that have to be considered.

A trivial dependency is given if an instance has to be aborted because of $abort(W, C)$ or $abort-workflows-of(R)$. Then, local control actions affecting activity nodes of the same instance become obsolete and thus do not have to be considered anymore w.r.t. this aborted workflow.

A more complex dependency is given if a workflow instance I is affected both by a $suspend(W, C)$ or $suspend-workflows-of(R)$ control action with valid time VT on one side, and a local control action ca with valid time VT^* on the other side. We distinguish the following two situations:

1. The $suspend(W, C)$ or $suspend-workflows-of(R)$ control action is triggered during the valid time of ca . This situation constitutes an *unexpected delay* of the workflow instance I adapted according to ca and has to be handled by workflow monitoring (see Chapter 9)
2. The local control action ca is triggered during the suspension of I (Figure 7-5). If ca is handled by reactive adaptation for I , nothing has to be done as one has to simply wait until the control flow reaches an activity node and to check whether it is affected by ca according to Section 7.4.3. However, if ca is handled by predictive adaptation for I , we have to transform

ca VALID-TIME VT^* to

$$ca \text{ VALID-TIME } VT^{**} \quad \text{with} \quad VT^{**} := [end(VT), end(VT^*)] \quad (xxvi)$$

for I , i.e., the remaining time of the suspension is subtracted from the “left” side of the valid time VT^* of ca . This is necessary because otherwise predictive adaptation would estimate that workflow part P_{VT^*} starting at the failure node set $FNS_{I, ca}$ (e.g., at nodes 2 and 5 in Figure 7-5) and corresponding to the valid time VT^* . Thus, nodes such as node 7 in Figure 7-5 would also be considered though they will not be reached anymore during VT^* due to the suspension. This has to be avoided, as the valid time VT^* assigned to ca only states that the adaptation shall be applied for an interval starting at the point in time when ca has been triggered until the end of VT^* .

Note that the valid time VT^{**} in (xxvi) may not fulfill the conventions 1 and 2 listed in 7.3. In particular, if the end of VT is conditional (i.e., specified by *Unless* or *Until*), the begin of VT^{**} also is conditional. However, this convention violation affects only a control action that has already been triggered, and furthermore affects only the suspended workflow instance. It does not affect a failure rule in the rule base for which these conventions have been defined. If the begin of VT^{**} is conditional, AGENTWORK only has to keep track of the point in time when the condition is met to start the adaptation process as for any other local control action that is handled by predictive adaptation.

Furthermore, if the end of VT^* and therefore the begin of VT^{**} is explicitly specified by a date or duration and thus is known beforehand (e.g., is already known at t_2 in Figure 7-5), the adaptation of I could already be performed *during* the suspension of I . For example, if ca is a $drop(A, C)$ control action and node 3 an A -node, then this node could be dropped from the control flow already during the suspension.

If VT covers VT^* totally and $VT^{**} = [end(VT), end(VT^*)]$ therefore is the empty interval, ca does not affect I at all. In particular, if $ca = add(A, C)$ then of course I cannot be considered as a candidate for executing the additional A -node.

7.6.3 Dynamic Dependencies between Local Control Actions

At execution time, also several dependencies and incompatibilities between local control actions may occur that cannot be detected at rule definition time. Similar to the situation at rule definition time described in Section 7.5, we distinguish dynamic *redundancy* and dynamic *incompatibility*.

7.6.3.1 Dynamic Redundancy

Any action redundancy described in 7.5.1.2 can also occur at execution time if two rules R_1, R_2 first are triggered simultaneously by non-overlapping events (i.e., are triggered independently by different data), and if second the control actions of R_1, R_2 fulfill the redundancy criteria listed in 7.5.1.2. Thus, the control action being redundant w.r.t. the other can be dropped from the set of currently valid control actions. It could be argued that such dynamic action redundancies could be detected already at rule definition time by checking for all rules pairs R_1, R_2 whether they would trigger redundant control actions in case they would be triggered simultaneously. However, this does not bring much benefit as a rule pair R_1, R_2 possibly leading to dynamic action redundancy should be allowed in the rule base. For example, for the two activity definitions

$A_1 := \text{Drug-Administration}[]$ and $A_2 := \text{Drug-Administration}[\text{drug} = \text{"ETOPOSID"}]$

let us assume two rules

R_1 : *WHEN* *serious-renal-disorder(P)*
 THEN *drop(A₁, P) VALID-TIME [now, now + (7, day)]*

R_2 : *WHEN* *serious-neuropathy(P)*
 THEN *drop(A₂, P) VALID-TIME [now, now + (4, day)]*.

R_1 states that *all* drug administrations have to be dropped for the next seven days when the patient suffers from a serious renal disorder, independently from the particular drug or drug dosage. R_2 states that *only* ETOPOSID administrations have to be dropped for the next four days when the patient suffers from a serious neuropathy. Thus, when R_1 and R_2 are triggered simultaneously the control action of R_2 is redundant w.r.t. the control action of R_1 and can be dropped. As in clinical practice both events do not occur simultaneously very often, it should be allowed to have these two rules together in a rule base. Rather, it is more appropriate to drop such a redundant control action at execution time, if the seldom situation occurs that a patient suffers both from a serious renal disorder *and* a serious neuropathy.

Note that the action redundancy definition of 7.5.1.2 also covers the situation that for example R_2 is triggered later than R_1 (e.g., two days later), but that its control action valid time still is entirely covered by the control action valid time of R_1 . In contrast to this, dynamic *partial* redundancy is not addressed by AGENTWORK (see 7.5.1.2), as it can become arbitrarily complex, and as the focus of this thesis is not on rule execution. Furthermore, note that *event* redundancy is entirely determined at rule definition time so that it has not to be discussed in the context of dynamic dependencies.

7.6.3.2 Dynamic Control Action Incompatibility

Any action incompatibility described in 7.5.2 can also occur at execution time if two rules R_1, R_2 first are triggered simultaneously by non-overlapping events (i.e., are triggered independently by different data), and if second the control actions of R_1, R_2 fulfill the incompatibility constraints listed in 7.5.2. For example, for the activity definition

$A := \text{Drug-Administration}[\text{drug} = \text{"DOXYCYCLIN"}]$

let us assume two rules

R_1 : *WHEN* *serious-renal-disorder(P)*
 THEN *drop(A, P) VALID-TIME [now, Until (NOT serious-renal-disorder(P))]*

R_2 : *WHEN* *bacterial-infection(P)*
 THEN *add-repetitively(A, (1, day), P) VALID-TIME [now, now + (2, week)]*

As usually a serious renal disorder *and* a bacterial infection do not occur together, it should be allowed to have these two rules together in a rule base. However, in case that a patient simultaneously shows a serious renal disorder *and* a bacterial infection it would be triggered to drop DOXYCYCLIN until the renal disorder is over *and* to administer it daily for the next two weeks. As both valid time intervals intersects as long as it holds *NOT serious-renal-disorder(P)*, these two control actions are incompatible. Thus, when rules with such incompatible control actions are triggered simultaneously, the user has to be informed and has to resolve the situation manually. In case of our sample rules R_1 and R_2 , the user could decide to administer some antibiotic having less renal side-effects than DOXYCYCLIN.

Note that at first glance dynamic incompatibilities could be avoided by extending the *WHEN* parts of the rules in a way that incompatible control actions with overlapping valid time intervals cannot be triggered simultaneously. For example, one could extend the *WHEN* part of R_2 as follows

R_2 : *WHEN* *bacterial-infection(P) AND NOT serious-renal-disorder(P)*
 THEN *add-repetitively(A, (1, day), P) VALID-TIME [now, now + (2, week)]*

Then, the control actions of R_1 and R_2 cannot be triggered with overlapping valid time intervals, so that a dynamic incompatibility is avoided. However, this mechanism requires that all constellations in which a rule shall not be triggered have to be explicitly specified, and furthermore depends on the control actions of other rules. Thus, it is more suitable to manually resolve such incompatibilities at rule execution time as described above.

Another type of dynamic incompatibility may occur between *case*-related control actions on one side, and *resource*-related control actions on the other side. This is because both control action types finally affect activity nodes, so that incompatibilities may occur that cannot be foreseen at rule definition time. For example, imagine that the following two control actions are triggered with overlapping valid time intervals :

drop-activities-of(R) and *add(A₂, C)*. (xxvii)

If R describes a resource that is needed to execute an A_2 -node that is executed for case C , then the control action constellation of (xxvii) has to be viewed as dynamically incompatible as it states that an A_2 -activity that has to be added for C but also has to be dropped.

The detection of such dynamic incompatibilities between case-related control actions and resource-related control actions can directly be derived from that between case-related control actions described in Section 7.5.2: When it is known for the resource-related control action which activity nodes with which activity definitions are affected, it is checked on the basis of Table 7-4 whether this results in incompatibility or not. For example, when it is known that *drop-activities-of(R)* in (xxvii) affects an A_1 -node and A_1 subsumes A_2 , this corresponds to pair (1,5) in Table 7-4 and thus means incompatibility. Analogously to dynamic incompatibilities between *case*-related control actions, this has to be resolved manually, e.g., by removing one of the two control actions.

7.7 Summary and Discussion

In this chapter, we described the global and local control actions supported by AGENTWORK. These control actions cover a broad range of control flow failures. In particular, they allow to abort or suspend entire workflows, or to drop, replace, postpone or add activities w.r.t. a running workflow. Triggering events may be case-related or resource-related events. From the experiences of the author made with medical domains such as hemato-oncology, the set of supported control actions can be viewed as sufficient, as it covers most of the relevant actions needed for treatment adaptation. It is the assumption, that the set of supported control actions is also sufficient for many non-medical domains such as banking or insurance business, as medical domains typically are more complex than most business domains. One particular strength of the control action approach described is the usage of activity or resource *patterns* which are based on an object-oriented data schema. By using such patterns, control actions can be defined on a high level of abstraction. In particular, control actions do not have to refer to the exact activity or resource definitions of activity nodes. Rather, an activity node is affected by a control action if its activity or resource definition matches the pattern of the control action. This gives the failure handling approach much flexibility when specifying which events induce which sort of workflow adaptation.

Of course, one could think of further useful control actions, such as a control action adding a conditional branching to a running workflow. For example, let us assume a cancer patient that periodically gets heart arrhythmia, and thus may not get heart-toxic drugs during such an arrhythmia phase. Then, it could make sense for workflows running for this patient to dynamically add a conditional branching “*If heart arrhythmia / If no heart arrhythmia*” before a chemotherapy. The “*If no heart arrhythmia*” path could consist of the heart-toxic drug administrations, the “*If heart arrhythmia*” path of alternative drug administrations. However, to keep failure handling controllable, this dynamic insertion of conditional elements is not supported. In particular, the handling of conditional elements such as OR-SPLIT or LOOP-END nodes has been identified as one of the central problems of workflow duration estimation (see Chapter 6), so that control actions themselves should not insert further conditional elements into a workflow. Furthermore, situations such as the heart arrhythmia can often also be handled by the supported control actions. For example, the usage of a drug not attacking the heart instead of a heart-toxic drug could also be handled by a *replace* control action triggered whenever the patient has such a heart arrhythmia.

Concerning rule integrity and dynamic control action dependencies, we have described mechanisms to cover at least the most important types of redundancy and incompatibility. Of course, this does not cover the broad range of more complex aspects, such as partial redundancy effects that may occur between rules (as sketched in 7.5.1). However, as the handling of these advanced rule integrity aspects has been largely addressed in the fields of active databases and artificial intelligence [BARALIS 1999, GABBAY ET AL. 1998, SMITH & KANDEL 1993], we do not discuss these aspects in detail in this thesis.

The following two chapters of this thesis will describe how the control actions introduced in this chapter will be transformed into structural workflow adaptation on the node and edge level (Chapter 8 and Chapter 9).

This chapter describes the adaptation operators that translate the local control actions of Chapter 7 into structural workflow adaptations, i.e., into adaptations of a workflow's node and edge set. These adaptation operators, which are used by the adaptation agent, are relevant both for reactive and predictive adaptation. The specific problems of predictive adaptation, namely the *order* in which adaptation operators shall be invoked if *multiple* control actions affect the same workflow, will be discussed in Chapter 9 (*Predictive Control Flow Adaptation*).

Chapter 8 is organized as follows: Section 8.1 introduces the specific design goals, basic assumptions, and definitions of the AGENTWORK adaptation operators. In Section 8.2 we describe the *control flow* adaptation operators which translate local control actions into structural control flow adaptations. Section 8.3 describes how the *data flow* is adapted after a control flow adaptation. The chapter concludes with a summary and discussion in Section 8.4.

8.1 Design Goals, Basic Assumptions, and Definitions

In this section we describe the design goals and basic assumptions of the structural adaptation operators in AGENTWORK (8.1.1-8.1.2). Furthermore, we introduce some useful definitions (8.1.3).

8.1.1 Design Goals

Beside the overall goal of translating local control actions into structural workflow changes in a way preserving the semantics of these control actions, structural adaptation in AGENTWORK is motivated by the following specific goals:

1. *Structural Consistency of Adaptations*

A structural adaptation operator such as an operator deleting or inserting a node has to meet the control and data flow constraints introduced in 5.3.5.3 and 5.3.6.4. This is of particular importance, as after an adaptation a workflow may be affected by subsequent control flow failures which may require further temporal estimations. As the estimation algorithms introduced in Chapter 6 assume that the estimated workflow meets all control and data flow constraints, it has to be guaranteed that no structural adaptation operator violates them.

2. *Logical Consistency of Adaptations*

In addition to structural consistency, an adaptation operator should not violate the “logical” consistency of a workflow. Logical consistency is typically expressed by logical constraints expressing object characteristics and relationships that have to be satisfied from the application point of view [BREWKA 1996]. As this thesis cannot aim at solving the problem of constraint satisfaction [JONSSON & LIBERATORE 1999] in general, we only introduce one type of workflow-oriented logical constraints, namely that of logical sequences. Such logical sequences define logical units of works that should not be split up by adaptations (see 8.1.3.1).

3. *Minimization of Execution Delay*

The adaptation of a workflow should delay the execution of a workflow as little as possible. In particular, replaced or added nodes should not delay the execution more than necessary. For example, the insertion of new nodes due to additive control actions should not result in an “unbalanced” AND-SPLIT/AND-JOIN block where the execution of one path takes significantly longer than the other path. As the nodes after the AND-JOIN node cannot be executed before both paths have been executed, such unbalanced parallel paths would delay the execution of the nodes after the AND-JOIN node more than necessary.

4. *Controlling Pull-In and Push-Out Effects*

Finally, it should be possible to control pull-in and push-out effects. We recall from 3.4.4 (*Adaptation Side-Effects*), that for a workflow part P_{VT} that is assumed to be executed during a valid time interval VT , *pull-in* effects denote effects where a structural workflow adaptation causes that nodes so far not belonging to P_{VT} become a member of P_{VT} (i.e., they are “pulled into” P_{VT}). In this context, those pull-in effects have to be carefully controlled where nodes that originally would have been executed *beyond* VT now are affected by a control action valid during VT after having been pulled into P_{VT} .

Analogously, *push-out* effects denote effects where a structural workflow adaptation causes that nodes so far belonging to P_{VT} are not anymore a member of P_{VT} (i.e., they are “pushed out” from P_{VT}). Analogously to pull-in effects, structural workflow adaptation should carefully control those push-out effects where a node n is affected by a control action ca valid during VT but is pushed out from P_{VT} by another control action before ca could have been applied to n .

As we will see in this chapter, the control flow operators themselves have only limited possibilities to support this goal, as only push-out effects can be minimized to a certain degree. A more general support of this goal is only possible by predictive adaptation which will be described in

Chapter 9.

Obviously, the listed goals show some dependencies or have a contrasting nature. For example, if goal 3 is supported this automatically means to reduce push-out effects (goal 4), as execution delays that may induce push-out effects are avoided. In contrast to this, the consideration of logical sequence constraints (goal 2) may induce push-out effects as logical sequences forbid to outsource a node from a sequence to a parallel path which would help to avoid push-out effects (goal 4). Therefore, a compromise between such contrasting goals has to be found.

8.1.2 Basic Assumptions

Beside the goals that shall be achieved, the design of the AGENTWORK structural adaptation operators is based on two central assumptions, namely first a limited number of simultaneously triggered control actions and second resource availability:

1. Limited Number of Simultaneously Triggered Control Actions

AGENTWORK assumes that “on the average“ a single workflow will not be affected by “many“ (i.e., more than 2 or 3) control actions simultaneously. Though a violation of this assumption does not mean that the adaptation operators described in this chapter become incorrect, it justifies the complexity of these adaptation operators, as they will not be needed “too often“. Thus, this assumption reduces the possibility of uncontrollable side-effects during workflow adaptation.

This assumption of a limited number of simultaneously triggered control actions is at least consistent with the author’s experiences made in the medical workflow application HEMATOWORK, but is assumed to be valid for many other workflow applications, too. The main rationale for it is an indirect argument: If this assumption would not hold for a particular application, this would mean that workflows are affected by control actions very often, and thus that workflow definitions have to be changed very often. However, this would be a general argument *against* the usage of workflow management for this application. Rather, other approaches such as rule-based or constraint-based treatment systems may be more suitable in this case.

2. Resource Availability

AGENTWORK assumes that whenever a workflow is adapted, the necessary resources (e.g., users, programs etc.) to execute the *adapted* workflow *without delay* are available (if compared with the workflow before the adaptation, and if the control flow allows this). In particular, it is assumed that when a new activity node is added into a new parallel path of an AND-SPLIT/AND-JOIN block, all the resources are available to execute the new AND-SPLIT/AND-JOIN block in a time being not longer than the execution duration of the old block without the new parallel path. This assumption is of particular importance as AGENTWORK generates new parallel paths for many control actions to avoid violations of goal 3 and thus to minimize push-out effects.

The rationale for this resource availability assumption is, that control flow failures typically are

caused by “alert” situation, such as that a medical drug has to be administered additionally to get an emerging medical crisis under control. Thus, AGENTWORK assumes that in such an alert situation also additional resources are provided to execute the adapted workflow without delay. If this assumption does *not* hold for a particular workflow and adaptation situation, this does not lead to any inconsistent state of the workflow and not to a failure of the adaptation operators themselves. Rather, it means that temporal estimations may have to be reevaluated, as for example the paths of an AND-SPLIT/AND-JOIN block cannot be executed in parallel due to resource limitations, and thus implicitly have to be “serialized” during execution. Thus, the resource availability assumption should be seen as a “working hypothesis” that motivates some of the design decision concerning structural adaptation operators.

3. Usage of Average Case Estimation

For the sake of simplicity, we assume that for all workflow estimations of this chapter average case estimation is used (as defined in 6.1.1). If an other estimation strategy is used, the definitions and algorithms in this chapter can be easily adapted in a straightforward manner.

8.1.3 Definitions

In this section we introduce some useful definitions, namely logical sequences, null nodes, and non-adaptable edges.

8.1.3.1 Logical Sequences

Logical sequences express activity node sequences that form logical units of work and therefore should not be violated by structural workflow adaptation, e.g., by dropping nodes from them or by inserting new nodes into them. For example, in a medical domain a logical sequence could consist of a cytostatic drug administration, a preceding examination checking whether there are any contra-indications concerning this drug, and a post examination checking whether there are any short-term toxicity effects caused by this drug. For this sequence, it should neither be allowed to drop one of the three activities, as then the sequence would be incomplete, nor to add other activities somewhere between these three activities, as this would split up the specific medical semantics of this sequence.

Formally, a sequence $A_1 \rightarrow A_2 \rightarrow \dots \rightarrow A_m$ of activity definitions ($A_i \in \text{Activity-Defs}$) is called *logical sequence* if for any sequence of activity nodes $S = n_1 \rightarrow n_2 \rightarrow \dots \rightarrow n_m$ with $NAM(n_i) = A_i$ it holds that neither a node n_i may be dropped from S , nor a new *activity* node, OR-SPLIT or LOOP-END node may be added to S (as by adding conditional elements such as OR-SPLIT or LOOP-END nodes it cannot be guaranteed anymore that the full sequence will be executed). AND-SPLIT, AND-JOIN, LOOP-START or communication nodes may be added as then the sequence S is still executed entirely.

Of course, for any logical sequence one may find activities that may be inserted into it without violating the application logics of this sequence. For example, concerning the drug administration sequence an *additional* pre-examination activity could be inserted after the already existing pre-

examination activity. Thus, AGENTWORK views logical sequences as “soft” constraints, and not as “hard” constraints. This means that whenever a logical sequence may be violated by an adaptation, the user is requested and can allow the violation of the logical sequence.

8.1.3.2 NULL Nodes

For synchronization purposes, some of the control flow operators introduced in Section 8.2 will insert so-called NULL-nodes into a workflow. Such a NULL node has the followings semantics:

- It has no activity semantics, i.e., no activity definition is assigned to it.
- It has duration zero. Thus, it can be ignored by the workflow estimation algorithms described in Chapter 6. If a NULL node is the source of a synchronization edge, it is handled like any other node which is a source of a synchronization edge. Therefore, we do not have to extend our workflow execution and estimation model described so far.

8.1.3.3 Non-Adaptable Edges

Some of the control flow operators of Section 8.2 will assign waiting conditions to control flow edges to express temporal constraints, e.g., to postpone node executions. Thus, it has to be avoided that subsequent adaptation operators for instance insert new nodes in a way violating these temporal constraints. For this, AGENTWORK allows to assign the label *Non-Adaptable* to a control flow edge. This label means that neither attributes of this edge may be changed anymore (such as the waiting condition), nor that a node may be “inserted” into this edge.

8.2 Control Flow Operators

In this section, we describe the control flow operators needed to translate control actions into structural workflow adaptations. For this, we first introduce two operators that drop nodes and change attribute values (8.2.1–8.2.2). Then, we describe an auxiliary operator which is able to generate new parallel paths (8.2.3). It is rather complex but provides an important functionality frequently used by the following operators, and allows to describe these following operators in a compact manner. After this, we introduce the operators that are able to add nodes, to replace activity definitions, and to postpone nodes (8.2.4–8.2.7). Many of these operators are based on the adaptation operators introduced in [REICHERT 2000]. However, they significantly enhance Reichert’s operators, e.g., by integrating temporal estimations to support goals 3 and 4.

Generally, a control flow operator is described in six steps: First, it is described how the operator changes the control flow without considering loops, synchronization edges and edges with waiting conditions (Section *Effect on Control Flow without Loops, Synchronization and Waiting Edges*). Second, it is discussed how to deal with nodes that are located in a loop (Section *Handling of Loops*). Third, we discuss how the operator deals with synchronization edges and edges with waiting conditions (Section *Handling of Synchronization and Waiting Edges*). Fourth, we briefly discuss the effects the operator may have on the data flow (Section *Side-Effects on Data Flow*). Fifth, as a control flow operator adapts *running* workflows, we describe the adaptations of the node and

edge *states* that may become necessary (Section *State Adaptations*). Sixth, if necessary we discuss further relevant aspects of the particular operator (Section *Further Aspects*).

Note that for reactive adaptation, the control flow operators described in this section are directly invoked whenever a node is affected by the corresponding control action. For predictive adaptation, a *higher-level algorithm* decides in which order these operators have to be invoked to control pull-in and push-out effects. This algorithm will be described in Chapter 9.

8.2.1 Operator for Node Dropping

For node dropping, AGENTWORK provides the control flow operator

cfop-drop-node(*n*: Integer)¹.

This operator takes as input the identifier *n* of an activity node to be dropped. It is invoked when a node *n* is affected by a *drop*(*A*, *C*) or *drop-activities-of*(*R*) control action according to 7.4.3.

The precondition for this operator is that *n* is not an element of a logical sequence. If this precondition is violated, the user manually has to specify whether

- *n* shall not be dropped,
- *n* shall be dropped nevertheless, or
- *n* and all nodes of the logical sequence shall be dropped.

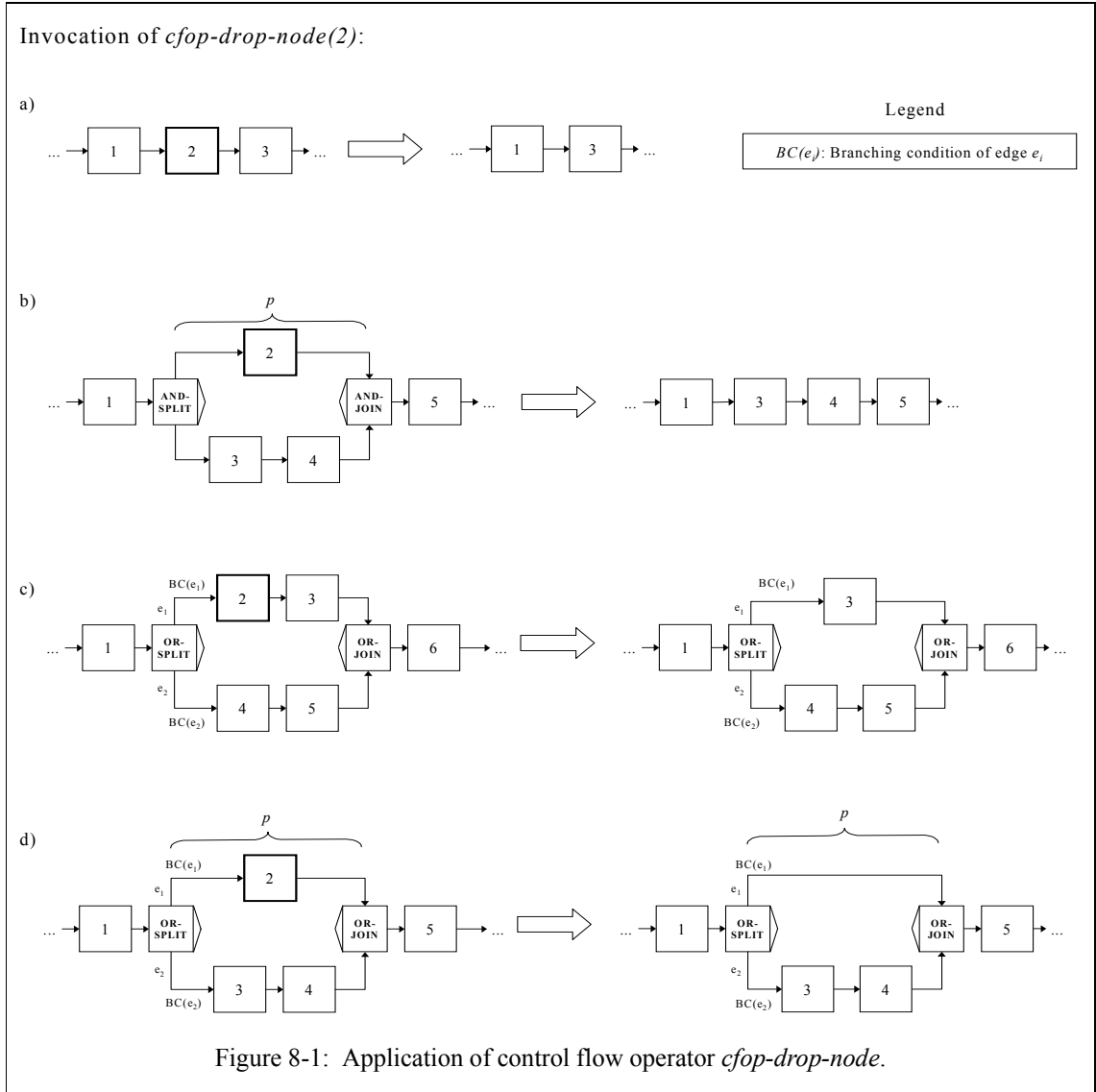
The latter alternative is suitable, if the other nodes of the logical sequence do not make much sense anymore if *n* is dropped. For example, if *n* represents a surgical intervention and the other nodes of the sequence represent pre- and post-examinations, it does not make much sense to execute only the examinations but not the surgical intervention.

8.2.1.1 Effect on Control Flow without Loops, Synchronization and Waiting Edges

The effect of this operator depends on the particular structure of the workflow part to which an affected node *n* belongs to. We distinguish the following cases (Figure 8-1):

- a) If *n* is located in a sequence of activity or communication nodes, *n* is simply removed from the control flow, and its predecessor and successor node are connected by a control flow edge (Figure 8-1 a).
- b) If *n* is the only node of a path *p* within an AND-SPLIT/AND-JOIN block, *p* is removed. If there is only one remaining path after *p* has been removed, the AND-SPLIT and the AND-JOIN node are removed as well, as they are not needed anymore (Figure 8-1 b).
- c) If *n* is the target node of an edge with branching condition, this branching condition has to be

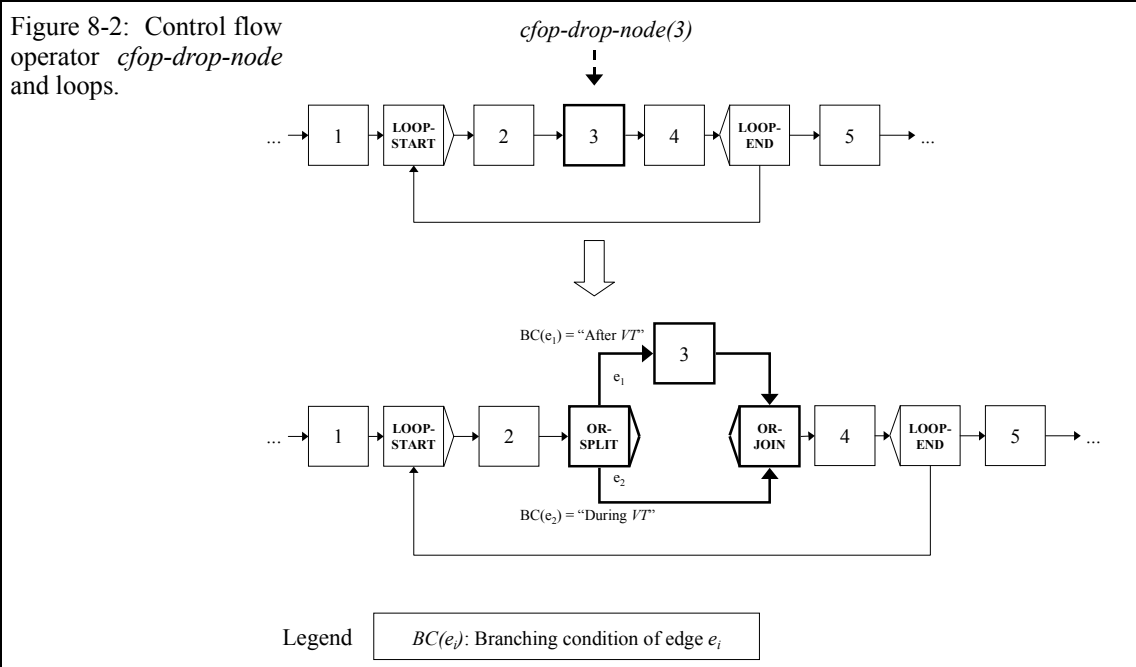
1. The prefix *cfop* stands for “control flow operator”



assigned to the edge connecting the direct predecessor and direct successor node of n (Figure 8-1 c). Note that n cannot be the *source* of an edge with branching condition, as this is only possible for OR-SPLIT or LOOP-END nodes (and as n is an activity node).

- d) If in c) n is the *only* node of a path p within an OR-SPLIT/OR-JOIN block, n is removed, but p is left within the block as empty path. This is necessary to keep the conditional semantics of the affected workflow part (Figure 8-1 d).

Figure 8-2: Control flow operator *cfop-drop-node* and loops.



Note that because parameter n always refers to an *activity* node, the application of *cfop-drop-node* never can drop single control flow nodes. Rather, only control flow node pairs may be dropped as shown in Figure 8-1 b). Thus, control flow constraint 3 (symmetrical blocks) introduced in Chapter 5 cannot be violated by this operator.

8.2.1.2 Handling of Loops

We now discuss how *cfop-drop-node* handles a node that belongs to a loop. This is a problem, as the semantics of a *drop(A,C)* or *drop-activities-of(R)* control action is that the execution of an affected node n shall be dropped *only* during the valid time VT assigned to the control action. As there may be loop iterations that occur outside VT we have to avoid that an execution of n is dropped for a loop iteration occurring beyond VT . There are two principal possibilities to deal with this problem:

1. Node n is removed from the loop sequence, and after the valid time has been elapsed, it is reinserted. However, this mechanism has the disadvantage that during the loop execution during VT it is not visible in the workflow that a node n will be executed when VT has expired. This may confuse users, as then suddenly a node is executed within the loop sequence which has not been executed and visible in former loop iterations.
2. An OR-SPLIT/OR-JOIN block with two conditional paths is inserted (Figure 8-2). One path is

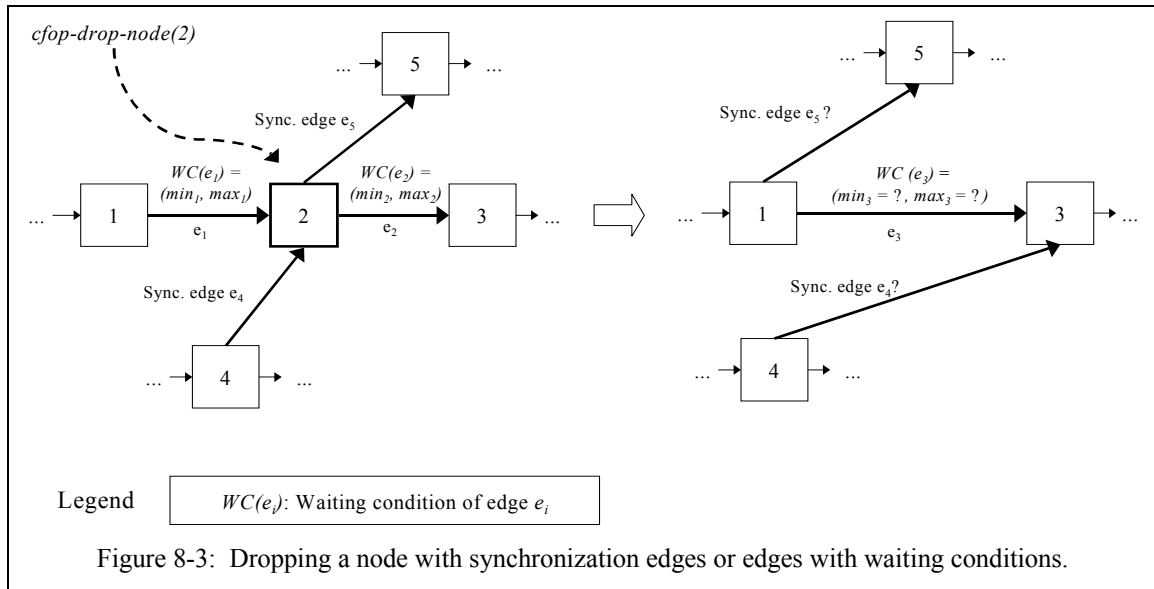
empty and is executed for the loop iterations during VT . The other path contains only n and is executed for the loop iterations beyond VT . This has the advantage that already during VT the user is aware of the fact that for loop iterations beyond VT node n will be executed (again). It has the disadvantage that it inserts an additional conditional branching that reduces readability of the workflow.

AGENTWORK favors the second mechanism as it shows the entire structure of the workflow to be executed, and does not temporarily “hide” nodes from the user. We omit a formal notation of the conditions $BC(e_1) = \text{“After } VT\text{”}$ and $BC(e_2) = \text{“During } VT\text{”}$ in Figure 8-2 as these are only technical terms comparing the system time with the end of VT for each loop iteration.

It could be argued that dropping a node by such a conditional branching as shown in Figure 8-2 could be used for *all* nodes to be dropped, not only for those belonging to a loop. This would also achieve that an affected node is not executed during VT , but is executed beyond VT . In particular, this would make complex predictive adaptation superfluous. However, this is not appropriate as such a mechanism is likely to produce unreadable workflows, as for every affected node a conditional branching would have to be inserted. If restricted to nodes in loops, it can be assumed that on the average this does not reduce readability too much.

8.2.1.3 Handling of Synchronization and Waiting Edges

We now describe how *cfop-drop-node* operates if the node n to be dropped is a source or target node of synchronization edges or edges with waiting conditions (Figure 8-3). Generally, AGENTWORK assumes that synchronization edges leading to or from n or waiting conditions on incoming or outgoing edges of n become superfluous when n is dropped. This is because a synchronization



edge or an edge with a waiting condition can be viewed as a matter only of the nodes directly connected by such an edge, and not of their successor or predecessor nodes. However, as this assumption does not need to be right for all workflow applications, AGENTWORK provides the configuration parameter

REMOVE-SYNC-AND-WAIT-EDGES-AFTER-NODE-DROPPING {YES, NO}

to specify the behavior of *cfop-drop-node* concerning synchronization edges or edges with waiting conditions. If this parameter is set YES, this means that for any dropped node *all* incoming or outgoing synchronization edges, and all waiting conditions of incoming or outgoing edges have to be dropped as well. If it is set NO, this means that for any dropped node, an authorized user has to be requested whether an incoming or outgoing synchronization edge has to be dropped or adjusted. For example, the user could specify that the outgoing synchronization edge e_5 in Figure 8-3 could start at the former predecessor node 1 of the dropped node 2. Furthermore, the user has to specify whether a waiting condition of an incoming or outgoing edge has to be dropped or has to be adjusted concerning the *min* or *max* values of the waiting conditions.

8.2.1.4 Side-Effects on Data Flow

After a node n has been dropped, all its incoming and outgoing data flow edges are dropped from the workflow as well. This may cause that an input object needed by another activity node or branching condition cannot be initialized anymore. The data flow adaptation that may become necessary then are described in 8.3.

8.2.1.5 State Adaptations

The state adaptations *cfop-drop-node* has to perform are shown in Figure 8-4. The “P” and “R” entries indicate whether a particular state constellation can occur for predictive (P) or reactive (R) adaptation. For example, a node in state *Untouched* can only be dropped through predictive adaptation as reactive adaptation by definition can only affect nodes which are at least in *Control-Activated* (see 7.4.3.1).

The matrices in Figure 8-4 have to be read as follows: The rows of the left state matrix show the possible states of the node to be dropped and its direct predecessor and direct successor node (and the connecting edges) before *cfop-drop-node* is invoked. The corresponding row of the right matrix shows the states of the direct predecessor and direct successor node after *cfop-drop-node* has been invoked (including the edge connecting them). For example, row 5 specifies that if node 2 in Figure 8-4 is in state *Data-Activated* and then dropped, the edge e_3 connecting the remaining nodes 1 and 3 has to be set to state *Control-Activated* (as this is the first control flow element of this path that has to be executed after the adaptation). Note that in row 3 the constellation that e_1 is in state *Committed* and node 2 in state *Untouched* (instead of *Control-Activated*) may occur if node 2 is the target of a synchronization edge and has not been set to state *Control-Activated* as this synchronization edge has not committed so far.

8.2.2 Operator for Changing Attribute Values

For changing attribute values, AGENTWORK provides the control flow operator

cfop-change-value(*n*: Integer, *p*: Object-Path, *f*: Function).

The first input parameter is the identifier *n* of a node affected by a *change-value*(*A*, *p*, *f*, *C*) control action. The second and third parameter specify the object path (e.g., attribute) value to be changed according to the function *f*. As this operator is trivial as it does not change the node or edge set of a workflow, there are no further details to be discussed.

8.2.3 Auxiliary Operator for New Parallel Paths

To minimize the temporal execution delay when adding, replacing, or postponing nodes, AGENTWORK frequently uses the technique of generating new parallel paths, which has first been introduced by [REICHERT & DADAM 1998]. For example, when a new node shall be added to a workflow due to an *add*(*A*, *C*) control action, one possibility to do this is to generate a new AND-SPLIT/AND-JOIN block with two paths. One path consists of the new node to be added, while the other path consists of a subsequence that already has been a part of the workflow before the *add*(*A*, *C*) control action has been triggered. Often, this has less temporal influence on a workflow than simply inserting the new node into an existing sequence. Another reason for generating a new AND-SPLIT/AND-JOIN block is that it then may not be necessary to split up logical sequences.

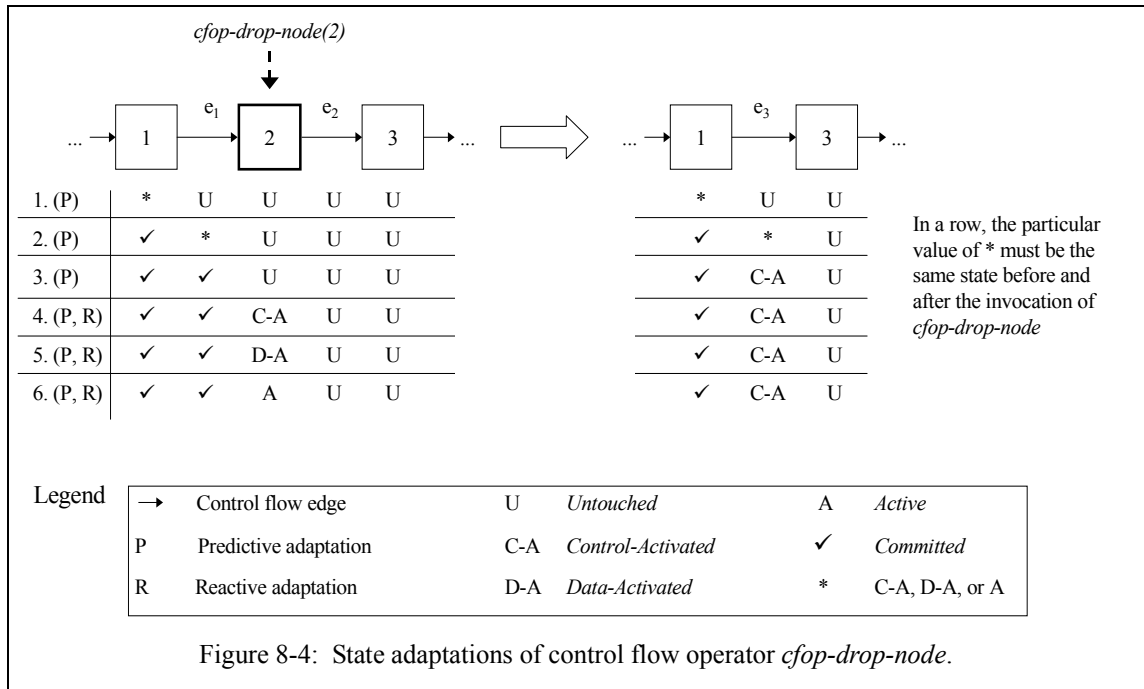
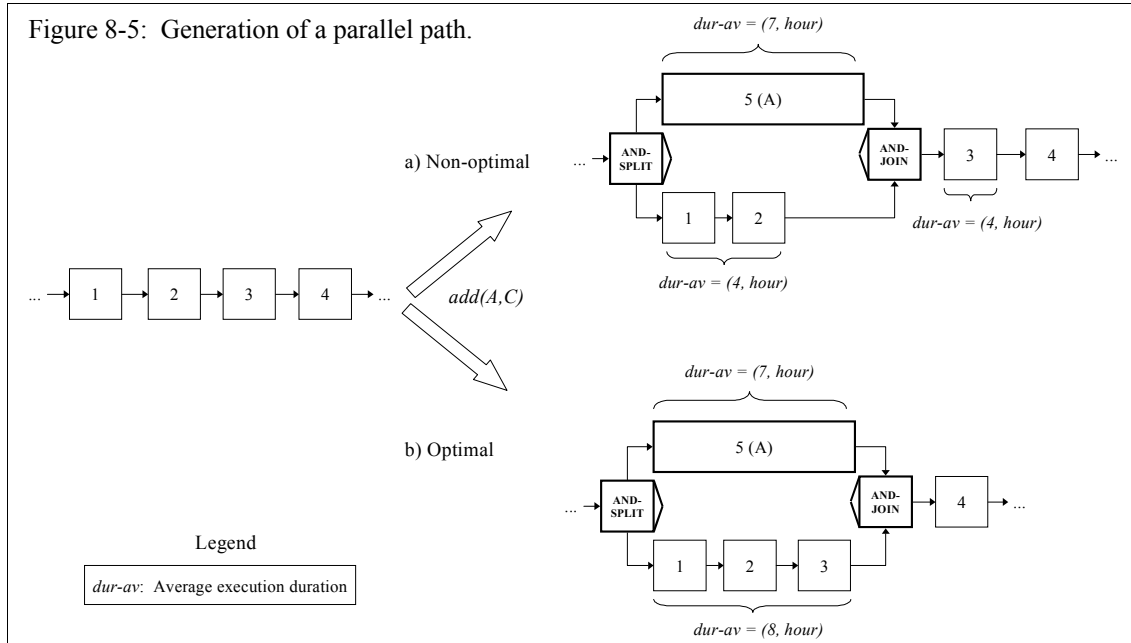


Figure 8-5: Generation of a parallel path.



However, to avoid an execution delay by the new AND-SPLIT/AND-JOIN block itself, one has to take care that the execution duration of the new parallel path is not longer than the execution duration of those nodes of the other parallel path that still have to be executed. Otherwise, the new AND-SPLIT/AND-JOIN block itself would delay the execution of successor nodes more than necessary. For example, in Figure 8-5 a) a new parallel path with an A -node has been added, due to an $add(A, C)$ control action. However, the average duration of this A -node is 7 hours while the average duration of node 1 and 2 together is only 4 hours. Thus, the execution of the A -node delays the duration of node 3 by 3 hours, if compared with the original sequence before the adaptation. This can be avoided by adding nodes to the shorter path, e.g., by also adding node 3 to the lower path in Figure 8-5 b). More generally, we can state the following criterion:

Criterion 8.1: Temporally optimal new AND-SPLIT/AND-JOIN blocks

A (new) AND-SPLIT/AND-JOIN block is called temporally optimal if the assumed average execution duration for the path of already existing nodes that still have to be executed (e.g., nodes 1, 2, and 3 in Figure 8-5 b) is at least as long as the average execution duration needed for the other parallel path generated to satisfy the triggering control action (e.g., for the path for the new A -node 5 in Figure 8-5 b).

The average execution duration needed for the path generated to satisfy the triggering control action is also called the *minimal* average execution duration of each path of the new AND-SPLIT/AND-JOIN block (e.g., in Figure 8-5 the minimal average execution duration for each path is 7 hours, which is only fulfilled for Figure 8-5 b).

Besides considering this temporal optimization aspect, the generation of a new AND-SPLIT/AND-JOIN block of course has to meet the control flow constraints described in 5.3.5.3 (*Control Flow Constraints*). In particular, due to control flow constraint 3 (symmetrical blocks) it is not allowed to add AND-SPLIT or AND-JOIN nodes at every position.

To generate such an AND-SPLIT/AND-JOIN block with two paths being temporally “balanced” in the sense of Criterion 8.1, AGENTWORK provides the control flow operator

cfop-gen-empty-parallel-path(*n*: Integer, *d*: Duration): {FALSE, TRUE}.

This operator takes as first input the identifier *n* of the node before which the AND-SPLIT node needed for the new parallel path shall be inserted. For example, *n* may be a node for which an added node shall be executed in parallel, so that the AND-SPLIT node has to be inserted before *n* (e.g., in Figure 8-5 it is *n* = 1). The second input *d* describes the minimal average execution duration according to Criterion 8.1. For example, in Figure 8-5 *cfop-gen-empty-parallel-path* would be invoked with *d* = (7, hour). The return value FALSE or TRUE indicates whether *cfop-gen-empty-parallel-path* has been able to generate the required new parallel path.

As preliminary remarks, two points are important to note:

- First, *cfop-gen-empty-parallel-path* obviously has to perform temporal estimations to check whether the temporal optimization criterion of Criterion 8.1 can be satisfied. This is not inconsistent to our remark at the beginning of this chapter, that all operators described are relevant both for predictive and reactive adaptation. This is because though reactive adaptation does not estimate which workflow part corresponds to the valid time of a control action, this adaptation strategy does not principally exclude that path durations are estimated as the missing of duration information or the existence of unresolvable conditions are only two of several possible reasons for using reactive adaptation. Other reasons for using reactive adaptation may include that the valid time interval of a triggering control action is conditional as described in Chapter 3 and Chapter 7, so that temporal estimations of path executions nevertheless may be possible.
- Second, due to the incorporated temporal estimations, *cfop-gen-empty-parallel-path* is a rather complex operator. However, once it has been defined, the operators that are able to add nodes, to replace activity definitions, and to postpone nodes can be introduced much more easily.

8.2.3.1 Effect on Control Flow without Loops, Synchronization and Waiting Edges

The operator *cfop-gen-empty-parallel-path* recursively searches for the minimal block (see 5.3.5.4) that allows to insert a new parallel path with a minimal average execution duration *d* of the resulting AND-SPLIT/AND-JOIN block. Table 8-1 contains the specification of the underlying algorithm. Illustrating examples are given in Figure 8-6. We make the following remarks:

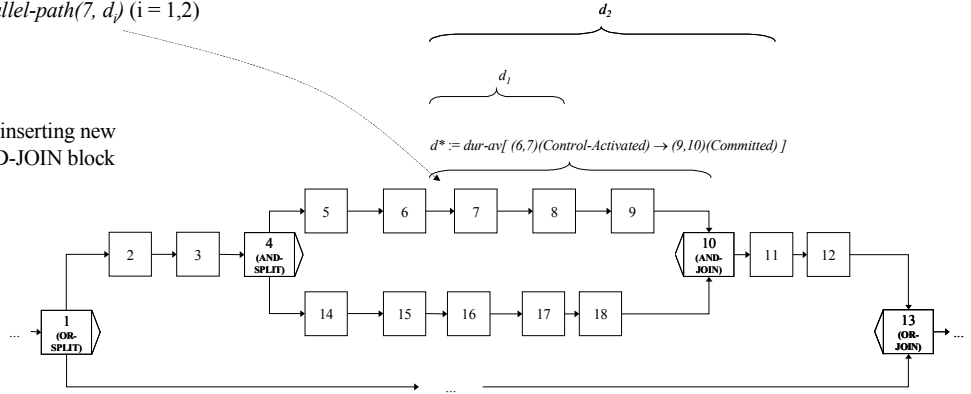
1. The further adaptation of the new AND-SPLIT/AND-JOIN block is subject of the other control flow operators that invoke *cfop-gen-empty-parallel-path*. For example, if a new node has to be added to the workflow due to some additive control action, this node will be inserted into the

Algorithm <i>cfop-gen-empty-parallel-path</i>		Examples (from Figure 8-6)	
	Input: n : Integer, d : Duration; Input: none Local variables: o, c, as, aj : Integer, // Node identifiers mb : (Integer, Integer), // Tuple identifying minimal block of a node d^* : Duration;	$n = 7$	
		$d = d_1$	$d = d_2$
1.	$o = n$;		
2.	Determine minimal block mb of n ;	$mb = (4, 10)$	
3.	Set c to closing node of mb ;	$c = 10$	
4.	Estimate $d^* = dur_{av}[e_1(Control-Activated) \rightarrow e_2(Committed)]$; (with e_1 being the incoming edge of n and e_2 being the incoming edge of c w.r.t. the path to which n belongs. The duration estimation has to start at e_1 and not at n , as e_1 may contain a waiting condition)	Estimate average time needed to reach node 10, i.e., until edge $e_2 = (9,10)$ commits after edge $e_1 = (6,7)$ has been set to state <i>Control-Activated</i>	
5.	IF estimation not possible { RETURN FALSE ; }		
6.	IF $d^* \geq d$ OR $c = \text{END node}$ { // sufficient time within mb or // end of workflow definition	True for $d = d_1$	False for $d = d_2$
7.	Insert new AND-SPLIT node as directly before o ;	For $d = d_1$ (Figure 8-6 b): Insert AND-SPLIT node $as = 19$ before 7	
8.	Insert new AND-JOIN node aj directly before c ;	Insert AND-JOIN node $aj = 20$ before 10	
9.	Insert empty path (i.e., new empty control flow edge) between as and aj ;	Insert edge (19,20)	
10.	Exit; }	Exit	
11.	ELSE $d^* < d$ { // no sufficient time within mb	False for $d = d_1$	True for $d = d_2$
12.	Set o to opening node of mb ;	$o = 4$	
13.	Set mb to minimal block of mb ;	$mb = (1,13)$	
14.	Go to step 3; } // next recursion step	For $d = d_2$ (Figure 8-6 c): Going to step 3 means: Insert AND-SPLIT node $as = 19$ before node 4; Insert AND-JOIN $aj = 20$ before node 13; Generate empty path between 19 and 20	
Recall from 5.3.5.4 that by definition the minimal block of a minimal block mb is not mb itself.			
		Recursion step 1 (for $d = d_1$ and $d = d_2$)	
		Recursion step 2 (for $d = d_2$)	

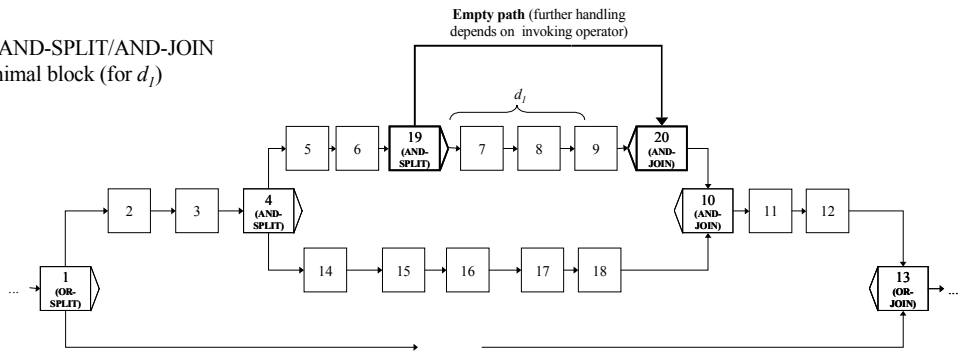
Table 8-1: Algorithm for generating an AND-SPLIT/AND-JOIN block with minimal average execution duration d .

cfop-gen-empty-parallel-path(7, d_i) ($i = 1, 2$)

a) Workflow before inserting new AND-SPLIT/AND-JOIN block



b) Inserting new AND-SPLIT/AND-JOIN block into minimal block (for d_1)



c) Inserting new AND-SPLIT/AND-JOIN block into enclosing minimal block (for d_2)

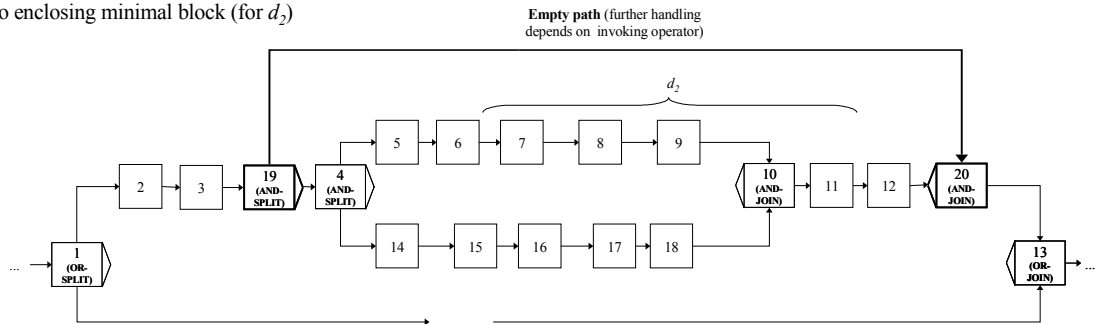


Figure 8-6: Application of control flow operator *cfop-gen-empty-parallel-path*.

generated empty parallel path (i.e., “into” the control flow edge between the AND-SPLIT and the AND-JOIN node).

2. If the end of the workflow is reached (i.e., $c = \text{END}$ node in line 6 of Table 8-1), the new AND-SPLIT/AND-JOIN block is then finally inserted, i.e., the AND-JOIN node is directly inserted before the END node. In this case the AND-SPLIT/AND-JOIN block may not be optimal in the sense of the temporal optimization Criterion 8.1. However, for $c = \text{END}$ node this is irrelevant as then the END node is the only node that may be delayed.
3. Note that the new AND-SPLIT node may be located in a workflow part that already has been executed, independently from the question whether *cfop-gen-empty-parallel-path* is invoked during reactive or predictive adaptation. At first glance, this may be irritating. However, it only requires that the workflow engine is able to notice and start parallel paths that dynamically have been added, even if the AND-SPLIT node has been inserted into a part already executed. The particular state setting of the new AND-SPLIT and its outgoing edges is described in 8.2.3.5 (*State Adaptations*).
4. If a branching condition BC has been assigned to the edge (l, m) with l, m being the two nodes between which the new AND-SPLIT is inserted (e.g., this would be the case if node 3 in Figure 8-6 c) would be an OR-SPLIT node)², BC is simply assigned to the edge $(l, \text{AND-SPLIT})$ while to the edge $(\text{AND-SPLIT}, m)$ no branching condition is assigned. This is appropriate, as the new AND-SPLIT node can simply be viewed as another node (without duration) of the sequence to which l and m belong.

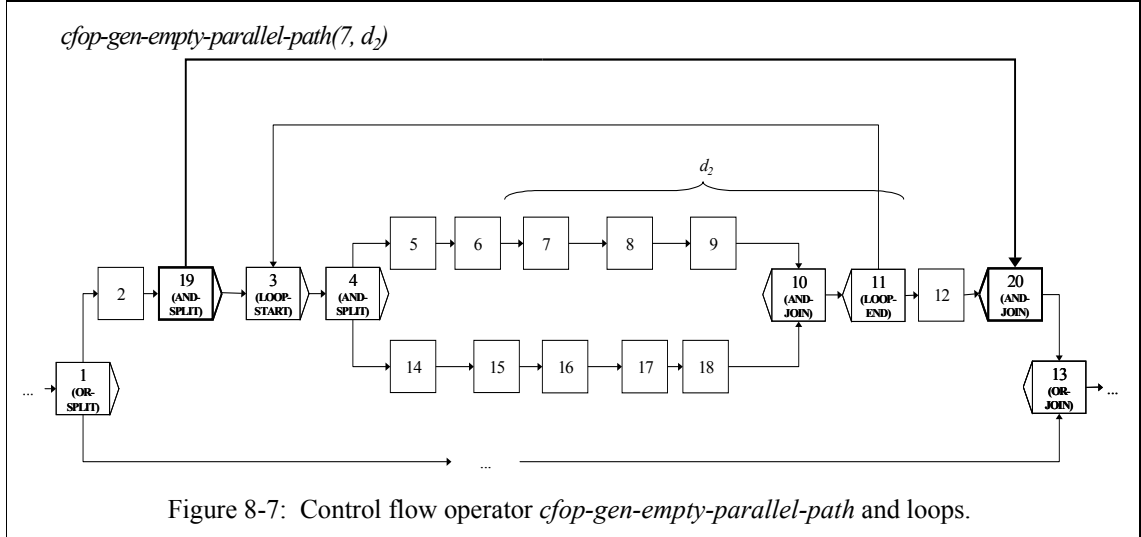
8.2.3.2 Handling of Loops

The algorithm of Table 8-1 also works if loops are contained in the workflow, as loops are only special blocks. For example, let us assume that the workflow of Figure 8-6 contains an additional loop (nodes 3 and 11 in Figure 8-7), and that *cfop-gen-empty-parallel-path* is invoked with the parameter d_2 from Figure 8-6. Then, an additional recursion step would be needed as it is not possible to find a path within $mb = (3, 11)$ that fulfills the duration condition in line 6 in Table 8-1. Thus, the new AND-SPLIT/AND-JOIN block is generated “around” this loop. The question whether an insertion of a new AND-SPLIT/AND-JOIN block *within* a loop is valid for *all* loop iterations or only for *some* of them is a matter of the operator invoking *cfop-gen-empty-parallel-path*, in particular of the valid time of the triggering control action.

8.2.3.3 Handling of Synchronization and Waiting Edges

Let l and m be the two nodes between which the new AND-SPLIT resp. AND-JOIN node is

2. Note that the new AND-JOIN node always is directly inserted before a closing block node, i.e., an already existing AND-JOIN or OR-JOIN node (see line 8 in Table 8-1). Thus, the two nodes between which the new AND-JOIN node is inserted can never be connected by a conditional edge, according to the workflow definition model of Chapter 5.



inserted. If l or m are the source or target of synchronization edges, these are kept without change. A waiting condition that has been assigned to the former control flow edge (l, m) is handled analogously to a branching condition (8.2.3.1), i.e., it is assigned to the edge $(l, \text{AND-SPLIT})$ resp. $(l, \text{AND-JOIN})$. For example, if a waiting condition would have been assigned to edge $(3, 4)$ in Figure 8-6 c), this waiting condition would be assigned to the edge $(3, 19)$.

8.2.3.4 Side Effects on Data Flow

The operator *cfop-gen-empty-parallel-path* has no effects on the data flow as it only inserts a new AND-SPLIT/AND-JOIN block and thus does not add or drop activity nodes.

8.2.3.5 State Adaptations

The state adaptations *cfop-gen-empty-parallel-path* has to perform are shown in Figure 8-8. For example, row 1 specifies that if node 1 is in state *Control-Activated*, *Data-Activated*, or *Active*, and if edges e_1 , e_2 and node 2 are in state *Untouched*, that then the new AND-SPLIT node and the new edges e_3 , e_4 have to be set to state *Untouched*. The new AND-JOIN node is omitted in Figure 8-8, as this node is always inserted into a workflow part not reached by the control flow at the moment of the insertion, so that this node is always set to state *Untouched*.

8.2.3.6 Further Aspects

Note that *cfop-gen-empty-parallel-path* always generates a *new* AND-SPLIT/AND-JOIN block, and does not check whether it could use already existing AND-SPLIT/AND-JOIN blocks to which only another parallel path has to be added. Such superfluous AND-SPLIT/AND-JOIN blocks can be merged together by reduction operators such as those introduced in [REICHERT 2000] to increase readability. Thus, we do not describe this sort of “syntactical optimization” here.

Figure 8-8: State adaptations of control flow operator *cfop-gen-empty-parallel-path*.

Legend

\rightarrow	Control flow edge	U	Untouched	A	Active
P	Predictive adaptation	C-A	Control-Activated	✓	Committed
R	Reactive adaptation	D-A	Data-Activated	*	C-A, D-A, or A

In this section, we describe the operator that adds a single node to a workflow if an *add(A, C)* control action has been triggered (the operator that handles an *add-repetitively(A, d, C)* control action will be described in 8.2.5). Note that it has already been discussed in 7.4.3.3 how AGENTWORK determines an appropriate workflow to which such a new node can be added.

$$cfop\text{-}add\text{-}node(A: Activity\text{-}Def, c: Case, n: Integer).$$

- If *cfop-add-node* is invoked during *reactive* adaptation, *n* may be any node of the failure node set of the triggering *add(A, C)* control action (see 7.4.3.3). This achieves that the new node is executed as soon as possible and thus is executed during the valid time of the triggering control action.

- If *cfop-add-node* is invoked during *predictive* adaptation, n may be any node of the workflow part P_{VT} corresponding to the valid time VT assigned to the triggering $add(A, C)$ control action. The question *which* node n within this workflow part P_{VT} shall be selected is a matter of a higher-level algorithm and will be discussed in Chapter 9.

8.2.4.1 Effect on Control Flow without Loops, Synchronization and Waiting Edges

Concerning the effect on the control flow, we distinguish two principal mechanisms of *cfop-add-node* to add a single A -node, namely *sequential* and *parallel* add.

Sequential add: The straightforward way to insert the new A -node is to insert it directly behind the node specified by the n -parameter (Figure 8-9).

The only problematic constellation for sequential add – if invoked during reactive adaptation – is that n is an OR-SPLIT node for which it is not known yet which of the conditional paths will be executed³. In this situation, it either has to be estimated which of the conditional paths will be executed (if possible), or it has to be waited until the conditional branching is executed and thus until it is known which conditional paths of n are executed definitely (so that the new A -node can be inserted into such a path as the first node).

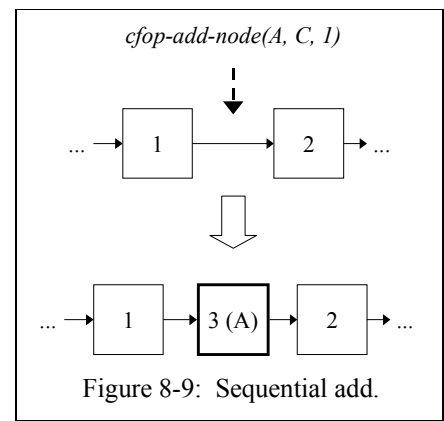


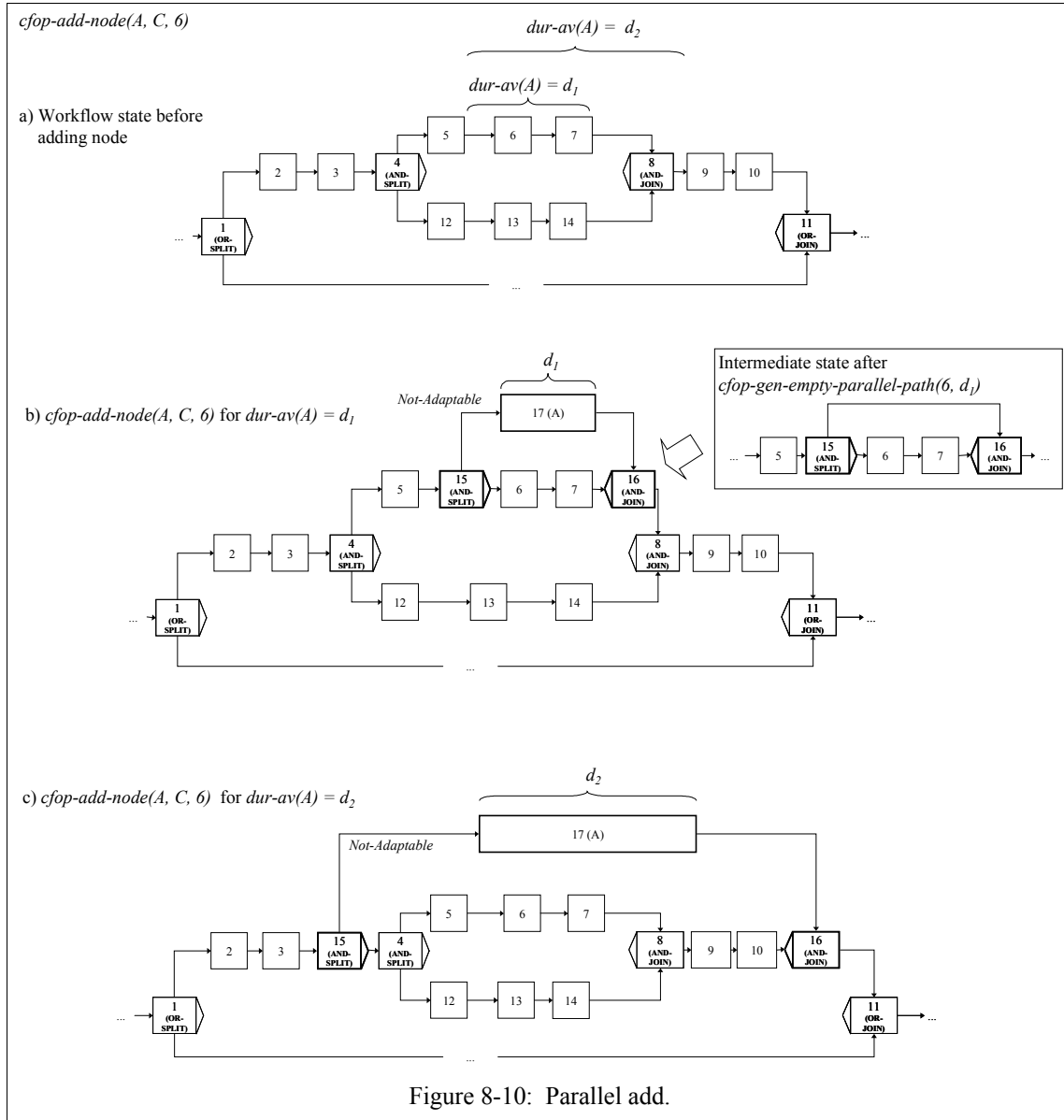
Figure 8-9: Sequential add.

A necessary condition for sequential add is that the insertion of a new A -node after node n does not violate a logical sequence (see 8.1.3.1). If a logical sequence is violated, an authorized user has to specify an alternative insertion point, i.e., a new value for n . AGENTWORK then checks whether the new value of n corresponds to a node of the failure node set (for reactive adaptation) or of P_{VT} (for predictive adaptation), and whether a sequential add at this new insertion area would violate any other logical sequence.

However, the main disadvantage of sequential add is that it may delay the execution of successor nodes of n (e.g., node 2 in Figure 8-9) more than necessary and thus may violate goal 3 (*Minimization of Execution Delay*). Therefore, *cfop-add-node* does not use sequential add as default strategy, but parallel add which is described now.

Parallel add: To avoid execution delays of successor nodes of n , *cfop-add-node* by default tries to insert the new node into a new parallel path. For this, *cfop-add-node* invokes *cfop-gen-empty-parallel-path* described in 8.2.3 to generate an AND-SPLIT/AND-JOIN block with an empty path to which the new node can be added. The parametrization of *cfop-gen-empty-parallel-path* is as follows:

3. Note that a failure node set may consist of only such an OR-SPLIT node so that we have to consider this constellation.



- The first parameter of *cfop-gen-empty-parallel-path* is set to n . For example, in Figure 8-10 *cfop-add-node* has been invoked with $n = 6$ so that *cfop-gen-empty-parallel-path* is invoked with $n = 6$, too.
- The second parameter d of *cfop-gen-empty-parallel-path* is set to the average duration of A , i.e.,

$d = \text{dur-av}(A)$, e.g., $d = d_1$ (Figure 8-10 b) or d_2 in (Figure 8-10 c).

If *cfop-gen-empty-parallel-path* is able to generate a new AND-SPLIT/AND-JOIN block with a new empty parallel path, *cfop-add-node* generates an *A*-node and inserts it into the new empty path. The edge between the new AND-SPLIT node and the new *A*-node is labeled as *Non-Adaptable* (see 8.1.3.3). This is done to avoid that for example further nodes are inserted between the new AND-SPLIT node and the new *A*-node, as this could cause that the *A*-node is not executed anymore during the valid time of the triggering control action.

In the example of Figure 8-10 b), the new *A*-node 17 is inserted into the empty path between the new AND-SPLIT 15 and the new AND-JOIN node 16. In the example of Figure 8-10 c), *cfop-gen-empty-parallel-path* has not been able to insert the new AND-SPLIT/AND-JOIN block into the minimal block of node $n = 6$, but only into the next surrounding minimal block.

Note that depending on the workflow execution stage, the new AND-SPLIT node may be inserted before a node which has already committed, e.g., if node 4 in Figure 8-10 c) has already committed before the adaptation. This is no problem. It only requires that *cfop-add-node* assigns the right states to the new *A*-node (i.e., assigns the state *Untouched* to the *A*-node) and to its incoming and outgoing edges to achieve a consistent execution semantics of the affected workflow part after the structural adaptation (see 8.2.4.5 for details on state adaptations). Furthermore, it requires that the AGENTWORK workflow engine can cope with this state constellation.

However, as a side-effect of inserting the new AND-SPLIT node before a node already committed, the new *A*-node may be executed *earlier* than node n , in particular it may not be executed in parallel anymore to node n . This is because due to the state adaptation matrix of Figure 8-8, the new AND-SPLIT node would be set to state *Committed*, and the edge between this new AND-SPLIT node and the new *A*-node would be set to state *Control-Activated* so that the new *A*-node may be executed directly after this. For example, in Figure 8-10 c) the new AND-SPLIT node 15 would be set to state *Committed*, and the edge (15,17) to state *Control-Activated*. Thus, *A*-node 17 may be started *before* node 6, e.g., if node 5 still is executed. However, there is no need to suppress this side-effect: The insertion area specified by node $n = 6$ is only a soft constraint and an *earlier* execution of the *A*-node never can violate the constraint that the *A*-node has to be executed during the valid time of the triggering control action.

If *cfop-gen-empty-parallel-path* is *not* able to generate a new AND-SPLIT/AND-JOIN block (i.e., returns *FALSE* in line 5 in Table 8-1), *cfop-add-node* performs a sequential add.

8.2.4.2 Handling of Loops

We now discuss what shall happen if *cfop-add-node* has inserted the new *A*-node into a loop. This is a problem, as the semantics of an *add(A,C)* control action is that an additional *A*-node shall be executed *exactly once* for case *C* during the valid time *VT* of the *add(A,C)* control action. If the new *A*-node has been inserted into a loop, it may be that the loop and thus the *A*-node is executed several times during *VT* which would violate the semantics of *add(A,C)*.

Analogously to *cfop-drop-node*, *cfop-add-node* handles this by inserting two conditional paths. The

first path contains the nodes of the loop sequence together with the additional A -node. The branching condition for this path is defined in a manner achieving that this path is executed only for the first loop iteration after the adaptation (we omit technical details here). The second path is executed for the remaining loop executions.

8.2.4.3 Handling of Synchronization and Waiting Edges

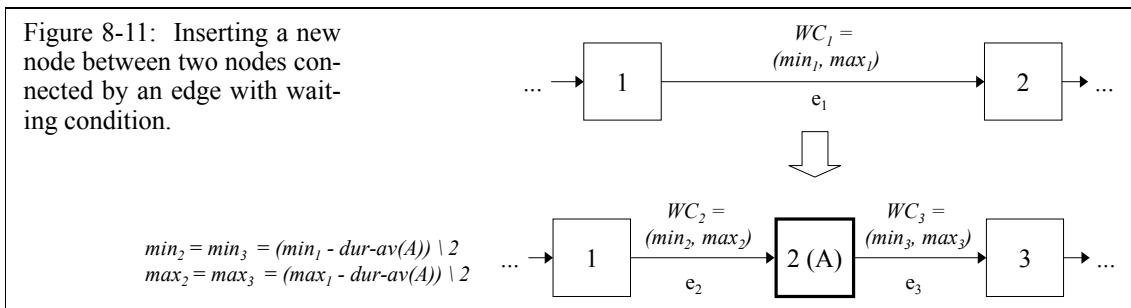
If *sequential* add is performed, the direct predecessor and successor node of the new A -node may be the source or target of synchronization edges or edges with waiting conditions⁴. In case of synchronization edges, these are kept without change. In case that the new A -node n shall be inserted between two nodes l and m with a waiting condition $WC((l,m)) = (min, max)$, AGENTWORK assumes that the waiting semantics of $WC((l,m))$ between l and m should be kept when n is inserted between l and m . For example, if l and m support drug administrations, a waiting condition taking into account some metabolism dependencies between them should not be violated when inserting a new node between l and m . This is handled by splitting $WC((l,m))$ up into two waiting conditions as follows (Figure 8-11): One waiting condition is assigned to the incoming edge of n , the other waiting condition is assigned to the outgoing edge of n . The *min* and *max* values of these two new waiting conditions are shown in Figure 8-11. Note that the average duration of n also has to be taken into account to achieve that the two new waiting conditions – together with the average duration of n – in the sum have the same waiting effect as the old waiting condition.

8.2.4.4 Side Effects on Data Flow

The operator *cfop-add-node* has no effects on *existing* data flow edges as it does not affect nodes already existing in the workflow, but only adds new nodes to a workflow. Nevertheless, due to the node adding, input objects that have not yet been considered by the data flow now have to be provided. The question how these required input objects can be provided is described in 8.3.

8.2.4.5 State Adaptations

The states that have to be assigned to the affected nodes by *cfop-add-node* are shown in

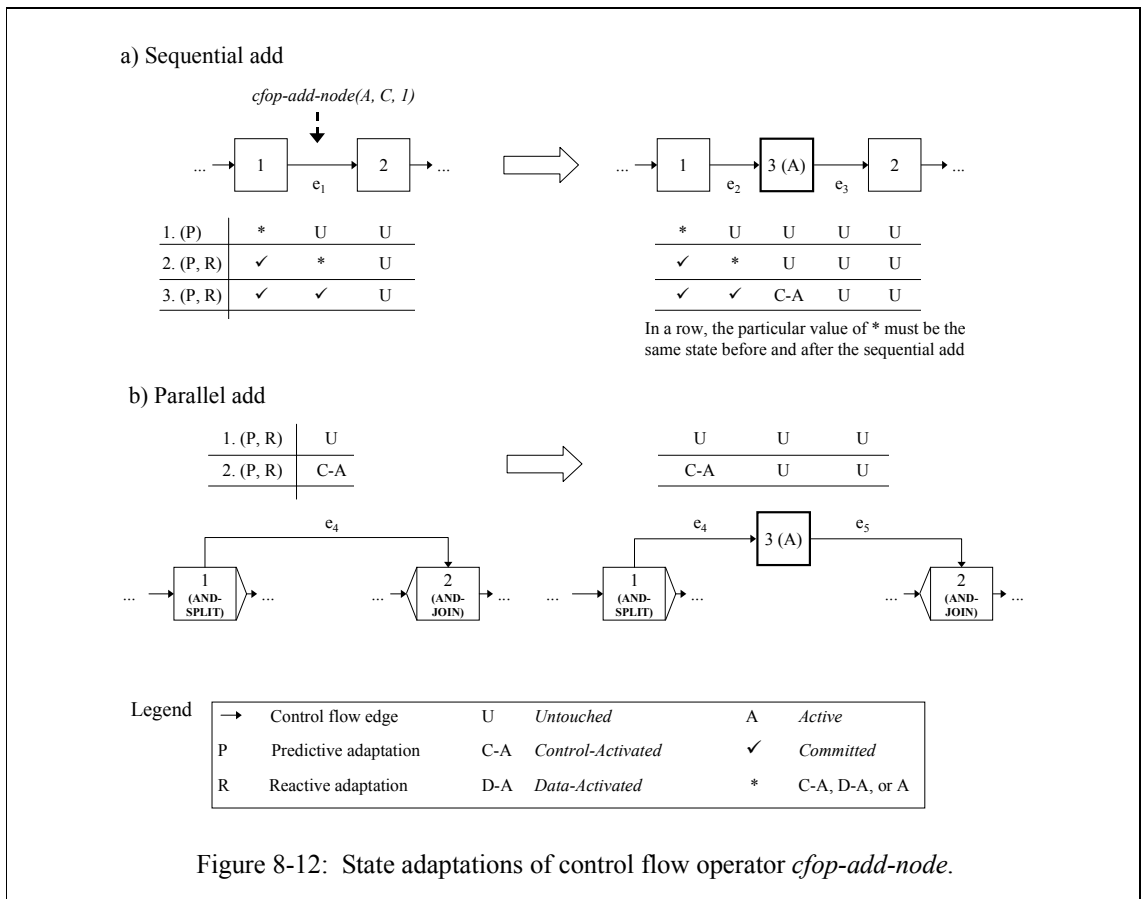


4. Note that *parallel* add inserts the new node into a newly generated path *without* synchronization edges or edges with waiting conditions.

Figure 8-12. For *sequential* add, the necessary state adaptations are shown in Figure 8-12 a). For *parallel* add, it has already been described in 8.2.3.5 which states have to be assigned to the new AND-SPLIT resp. AND-JOIN node, and the preliminary control flow edge e_4 connecting them. Thus, in Figure 8-12 b) we only have to specify the state that has to be assigned to the new A -node in dependency on the state of e_4 (if we view e_4 as the edge connecting the new AND-SPLIT node with the new A -node after *cfop-add-node* has been invoked). Recall that the new AND-JOIN node is always set to state *Untouched*, so that we do not have to consider the states of this node further.

8.2.4.6 Further Aspects

As *cfop-add-node* is triggered by an *add(A, C)* control action, we finally have to assign the case described by the control action parameter C to the new A -node. From the operational view, this is simply done by setting the return value of the function *case(x)* (5.4.3.5) to C (assuming that the new A -node has the node identifier x).



8.2.5 Operator for Repetitive Node Adding

To add nodes for repetitive activity executions, AGENTWORK provides the control flow operator

cfop-add-node-loop(*A*: Activity-Def, *p*: Duration, *cond*: Condition, *C*: Case).

The first parameter specifies the activity definition of the node to be executed repetitively. The second parameter specifies the period of the loop, i.e., the duration between two subsequent executions of the *A*-node. The third parameter specifies the termination condition of the loop. The fourth parameter specifies the *Case* object for which the new activity node shall be executed. The operator has no preconditions. It is triggered by an *add-repetitively*(*A,d,C*) control action.

8.2.5.1 Effect on Control Flow without Loops, Synchronization and Waiting Edges

Principally, *cfop-add-node-loop* realizes the repetitive execution of an *A*-node by inserting a loop with an *A*-node and a termination *cond* into the workflow. However, in contrast to *cfop-add-node*, *cfop-add-node-loop* does not try to add this loop into a new AND-SPLIT/AND-JOIN block being temporally optimized according to Criterion 8.1. This is because for a temporally optimized AND-SPLIT/AND-JOIN it would be necessary to estimate the execution of this loop. As already discussed in 6.4.4, the estimation of a loop's execution duration generally will be very imprecise, in particular when the loop is terminated by a qualitative termination condition such as “until leukocyte count higher than 2500” (in contrast to this, for *cfop-add-node* only the duration corresponding to a *single A*-node execution has to be estimated). Therefore, the easiest way to insert the loop into the workflow without causing any temporal delay is to instruct *cfop-gen-empty-parallel-path* to generate a “maximal” AND-SPLIT/AND-JOIN node, i.e., to insert the new AND-SPLIT node directly after the START node and the new AND-JOIN node directly before the END node, as shown in Figure 8-13. After this, *cfop-add-node-loop* generates a loop of the structure

LOOP-START → *A*-node → LOOP-END

and inserts it into the new empty path. The period of the loop specified by *p* is translated into a waiting condition with $min = max = p$ which is assigned to the edge between the LOOP-END node and the LOOP-START node. The termination condition specified by parameter *cond* is assigned to the edge between the LOOP-END node and the new AND-JOIN node. The edge between the new AND-SPLIT node and the new LOOP-START node is labeled as *Non-Adaptable*. The reason for this is the same as for *cfop-add-node*.

This way of inserting the loop with the new *A*-node never can cause a push-out effect as the only node that may be delayed is the END node which is irrelevant.

8.2.5.2 Handling of Loops

As *cfop-add-node-loop* always inserts the new AND-SPLIT directly after the START node and the new AND-JOIN directly before the END node, the new loop never can be a part of a higher-level

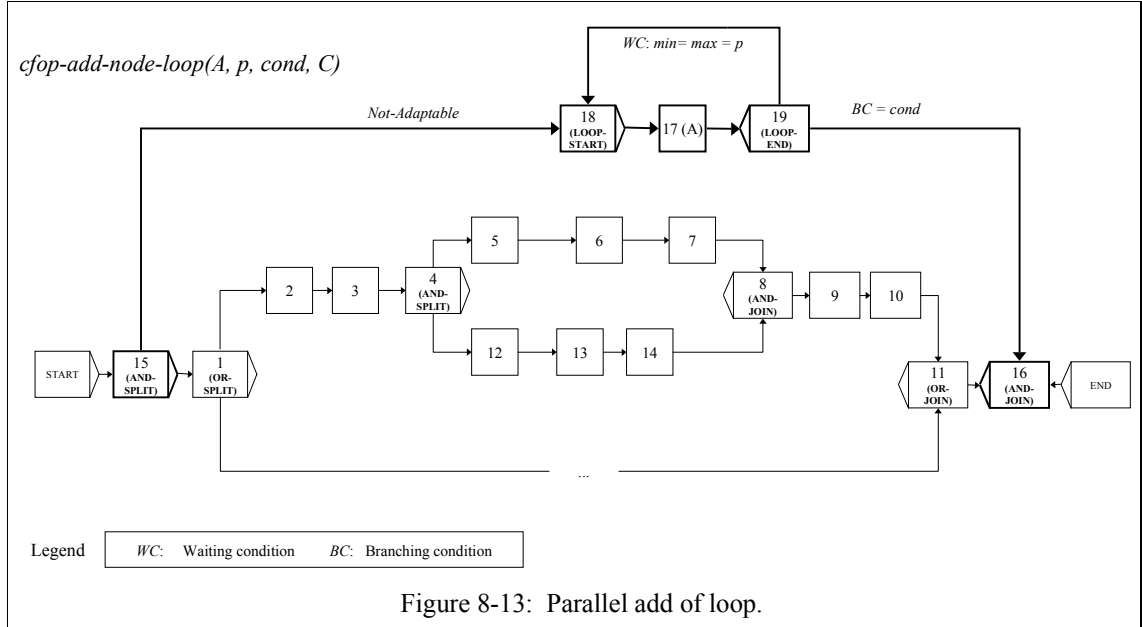


Figure 8-13: Parallel add of loop.

loop. Therefore, we do not have to discuss this problem for *cfop-add-node-loop*.

8.2.5.3 State Adaptations

The state adaptations that have to be performed by *cfop-add-node-loop* are analogously to that performed by *cfop-add-node* for parallel add. Thus, in Figure 8-12 b) the *A*-node has to be replaced by the loop LOOP-START → *A*-node → LOOP-END. To all three nodes of this loop, the state *Untouched* is assigned for row 1 and 2 in Figure 8-12 b).

8.2.5.4 Further Aspects

For further aspects – such as the side-effects on data flow and the case assignment – the statements made for *cfop-add-node* in the context of parallel add hold analogously for *cfop-add-node-loop*.

8.2.6 Operator for Replacing Activity Definitions

For replacing activity definitions, AGENTWORK provides the control flow operator

cfop-replace-act-def(*n*: Integer, *B*: Activity-Def),

which is invoked if a *replace*(*A*,*B*,*C*) control action has been triggered. This operator takes as its first input parameter the identifier *n* of the *A*-node for which the activity definition shall be replaced (in the following, *A* will always denote the *old* activity definition assigned to *n*). The second input parameter *B* is the new activity definition.

The precondition for this operator is that n is not an element of a logical sequence which would be violated by replacing the activity definition of n . If this precondition is violated, an authorized user has to specify whether the logical sequence may be violated (by replacing the activity definition of n) or not. If the latter possibility is selected, *cfop-replace-act-def* is not applied to n .

8.2.6.1 Effect on Control Flow without Loops, Synchronization and Waiting Edges

The effect of this operator on the control flow depends on the relationship between the durations of A and B . This duration relationship has to be considered to avoid that the replacing of an activity definition leads to a significant delay of the affected workflow and thus to a violation of goal 3 (minimization of execution delay). We distinguish between *sequential* and *parallel* replace.

Sequential replace: This type of replace is performed if the duration of a B -node is *not* significantly longer than the duration of an A -node. Then, *cfop-replace-act-def* simply performs a switch of the activity definition of n , i.e., $NAM(n)$ is set to B , with $NAM(n)$ being the function assigning an activity definition to an activity node (5.3.9). The position of n within the control flow is left unchanged (Figure 8-14).

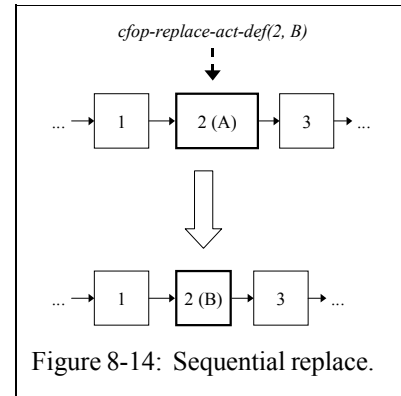


Figure 8-14: Sequential replace.

Parallel replace: This type of replace is performed if the duration of a B -node is significant longer than the duration of an A -node. Thus, a sequential replace as shown in Figure 8-14 may lead to a significantly delay of the affected path. Therefore, *cfop-gen-empty-parallel-path* is instructed to generate a new AND-SPLIT/AND-JOIN block with an empty path to which n with its new activity definition B can be moved to. The parametrization of *cfop-gen-empty-parallel-path* is as follows:

- The first parameter is set to the successor node of the node n by which *cfop-replace-act-def* has been invoked⁵. For example, in Figure 8-15 *cfop-replace-act-def* is invoked with node 7 so that *cfop-gen-empty-parallel-path* has to be invoked with node 8. We have to take the *successor* node as in order to achieve a temporally optimal AND-SPLIT/AND-JOIN block according to Criterion 8.1, *cfop-gen-empty-parallel-path* has to estimate the execution duration of the path starting at the old position of n but without the execution duration of node n (as n is sourced out from this path by parallel replace).
- The second parameter d is set to the average duration of B , i.e., $d = dur_{av}(B)$.

If *cfop-gen-empty-parallel-path* is able to generate the new AND-SPLIT/AND-JOIN block, node n with its new activity definition B is sourced out to the new parallel path. In particular, *cfop-drop-node* is invoked to drop node n from its original path. For example, in Figure 8-15 b) node 7 with its

5. As n is an activity node, it always has a non-ambiguous successor node.

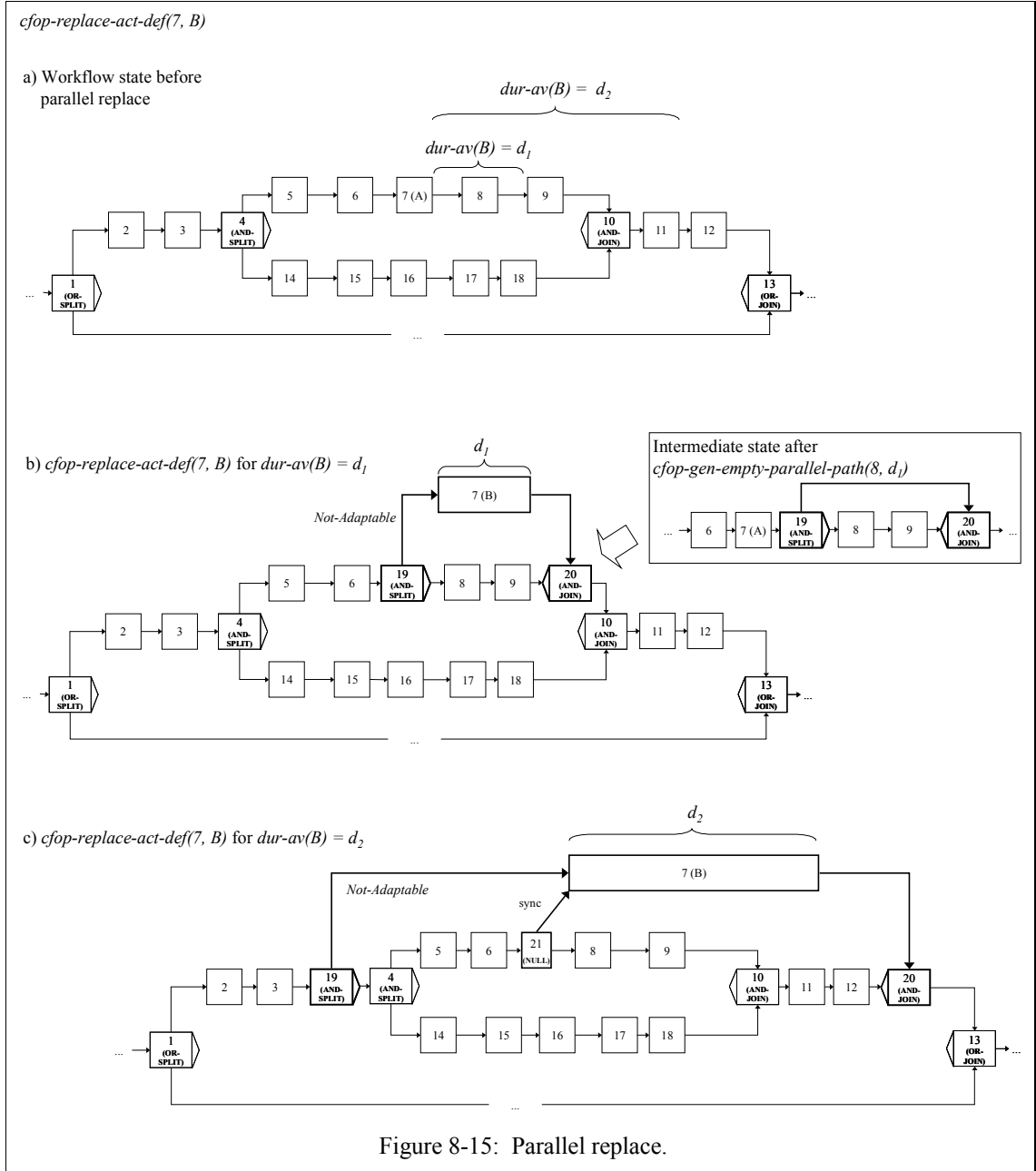


Figure 8-15: Parallel replace.

new activity definition B is dropped from its original sequence and added to the new path parallel to nodes 8 and 9. In the example of Figure 8-15 c), *cfop-gen-empty-parallel-path* has not been able to

insert the new AND-SPLIT/AND-JOIN block into the minimal block of node 8, but only into the next surrounding minimal block.

Note that depending on the workflow execution stage, the new AND-SPLIT node may be inserted before a node which has already committed, e.g., if node 4 in Figure 8-15 c) has already committed before the adaptation. Analogously to parallel add described in 8.2.4.1, this may have the consequence that n may be executed significantly earlier *after* the activity replacement (if compared with the situation *before* the activity replacement). While this side-effect of an earlier execution has been viewed as acceptable for added nodes (as there are no temporal relationships with already existing nodes that would have to be considered), the question is whether this side-effect should be allowed for activity replacement. For example, if the type of a diagnostic examination is replaced, it usually will be uncritical to execute the new examination earlier than the original one before the activity definition replacement. However, if for example the activity definition means that a different *drug* is applied, an uncontrolled earlier execution of the new drug may be fatal as implicit temporal relationships may be violated. Thus, as it is difficult to decide in general whether an earlier execution of n should be allowed or not, AGENTWORK requests the user. If the user wants to avoid that the node n for which the activity definition has been replaced is executed earlier, n is synchronized with its former position. Note that it is not suitable to simply synchronize n with an already existing activity node m (e.g., to synchronize node 7 with node 6 in Figure 8-15 c), as m may be affected (e.g., dropped) by subsequent adaptations. Therefore, a NULL node (see 8.1.3.2) is inserted at the former position of n , and n is synchronized with this NULL node by inserting the synchronization edge

$$((NULL\ node, Committed), (n, Control-Activated)) \quad (i)$$

to the workflow. For example, in Figure 8-15 c) a NULL node 21 is inserted between node 6 and 8, and node 7 is synchronized with it by an synchronization edge of form (i).

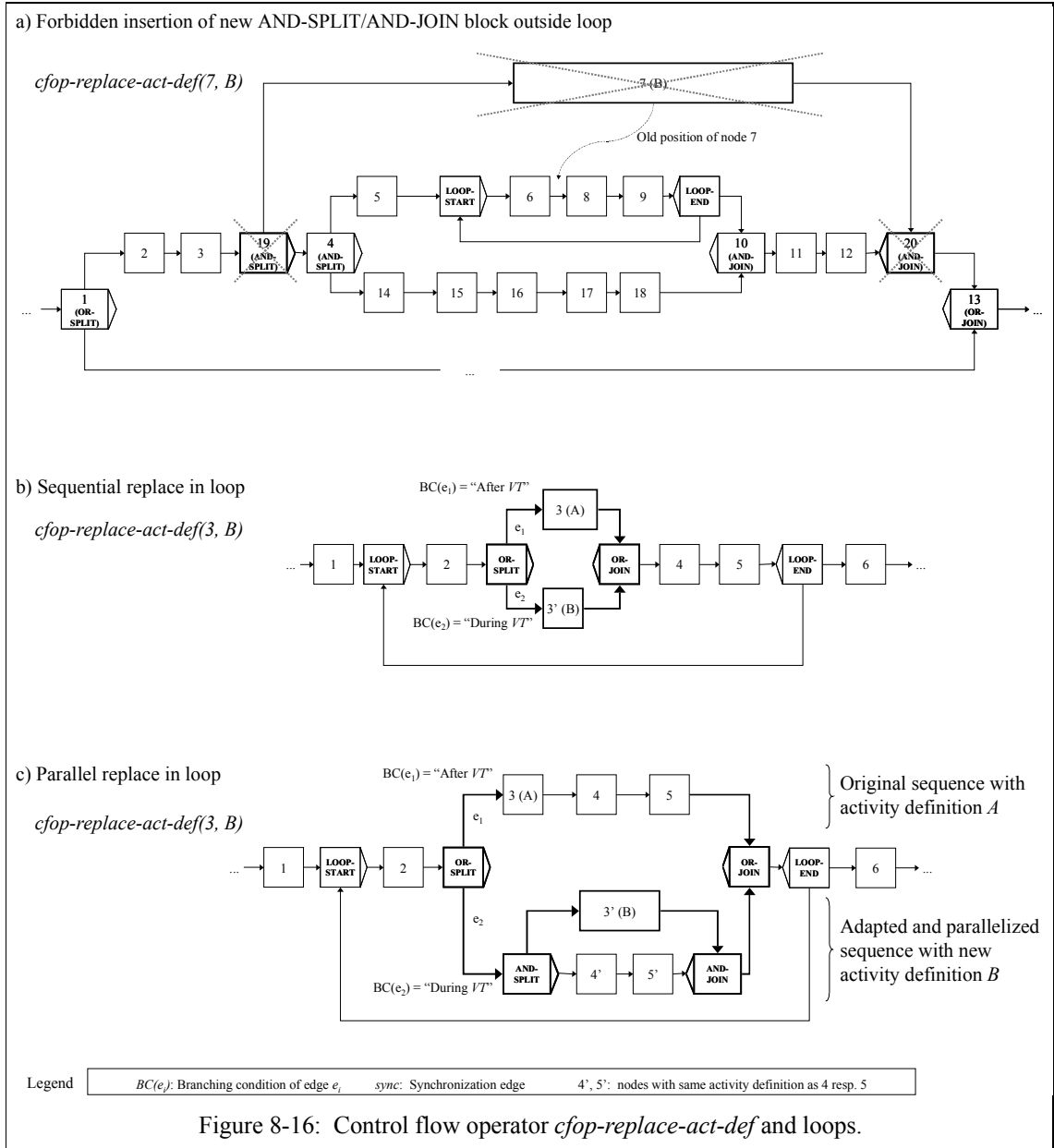
If the user agrees that the node for which the activity definition has been replaced is executed earlier, the control flow is left unchanged after node n has been sourced out to the parallel path, i.e., no NULL node and no synchronization edge is inserted.

If *cfop-gen-empty-parallel-path* is *not* able to generate a new AND-SPLIT/AND-JOIN block, *cfop-replace-act-def* performs a sequential replace as shown in Figure 8-14.

8.2.6.2 Handling of Loops

We now discuss what shall happen if the node n for which the activity definition has to be replaced belongs to a loop. Two problems have to be considered:

Problem 1: First of all, if parallel replace is performed the new AND-SPLIT/AND-JOIN block must not be inserted outside the loop. Otherwise, the node n would be executed more than once *before* the activity definition replacement, but only once *after* the activity definition replacement (Figure 8-16 a). Of course, this has to be avoided. Therefore, if n belongs to a loop *cfop-gen-empty-parallel-path* has to terminate its control flow exploration when detecting a LOOP-START node, and has to insert the AND-SPLIT and AND-JOIN node into this loop block.⁶



- We omitted this special case in the description of $cfop\text{-}gen\text{-}empty\text{-}parallel\text{-}path$ in Table 8-1 to concentrate on the core structure of the algorithm. Nevertheless, $cfop\text{-}gen\text{-}empty\text{-}parallel\text{-}path$ can be easily extended to cope with this loop case if invoked by $cfop\text{-}replace\text{-}act\text{-}def$.

Problem 2: Furthermore, we have to avoid that the activity definition is replaced for a loop iteration that is executed *after* the valid time VT of the triggering control action. Analogously to node dropping, this is handled by adding a conditional branching that checks whether VT has expired or not. The arguments for this are the same as given for *cfop-drop-node* in 8.2.1.2. This conditional branching is done for sequential replace (Figure 8-16 b) and for parallel replace (Figure 8-16 c).

8.2.6.3 Handling of Synchronization and Waiting Edges

We now discuss how *cfop-replace-act-def* shall behave if the node n of which the activity definition is replaced is a source or target node of synchronization edges or edges with waiting conditions.

There is no general solution for synchronization edges and edges with waiting conditions. For example, if one drug is replaced by a similar drug of the same substance group, it may be suitable to maintain all synchronization edges or waiting conditions as the temporal dependencies to other nodes are still the same. However, if one drug is replaced by a drug with the same indication but a totally different metabolism behavior, it may be that the temporal dependencies may change significantly. Therefore, AGENTWORK provides the configuration parameter

KEEP-SYN-AND-WAIT-EDGES-AFTER-ACT-DEF-REPLACE {YES, NO}

to specify the operator's behavior concerning synchronization edges or edges with waiting conditions. If it is set YES, this means that for any node for which the activity definition is changed, all incoming or outgoing synchronization edges, and all waiting conditions of incoming or outgoing edges are kept. If it is set NO, this means that for any node for which the activity definition is replaced, a user has to be requested whether an incoming or outgoing synchronization edge has to be dropped or kept, and whether a waiting condition of an incoming or outgoing edge has to be dropped, kept, or adjusted concerning the *min* or *max* values of the waiting condition.

8.2.6.4 Side-Effects on Data Flow

As the replacement of an activity definition means that also the input and output objects change, all incoming and outgoing data flow edges of the node with the replaced activity definition are also dropped. On one side, this may cause that an input object needed by another activity node or branching condition cannot be initialized anymore. On the other side, due to the new activity definition of the node, input objects that have not yet been considered by the data flow now have to be provided. The data flow adaptation that therefore may become necessary are described in 8.3.

8.2.6.5 State Adaptations

We now describe the state adaptations that have to be performed for a node n for which the activity definition has been replaced. For *sequential* replace, the relevant state adaptations are shown in Figure 8-17. Note that in general it is *not* possible to set n to the same state it has been in *before* the sequential replace. For example, during reactive adaptation the node n for which the activity definition has to be changed may already be in state *Data-Activated* (see 7.4.3.1). As the input object patterns may change because of the activity definition replacement, n has to be set back to state

Control-Activated, so that the new data can be retrieved by the data flow.

For *parallel* replace, the necessary state adaptations can be derived from those performed by the operators *cfop-gen-empty-parallel-path*, *cfop-drop-node*, and *cfop-add-node*: The state adaptations that have to be performed after the generation of the new AND-SPLIT/AND-JOIN block (little rectangle in Figure 8-16 b) have already been described in 8.2.3.5. The state adaptations that become necessary after *n* has been dropped by *cfop-drop-node* from its old location are described in 8.2.1.5. The state adaptations that become necessary after *n* has been inserted into the new path and after a NULL node has been inserted are described in 8.2.4.5. The synchronization edge between the NULL node and *n* always is set to state *Untouched*.

8.2.7 Operator for Node Postponement

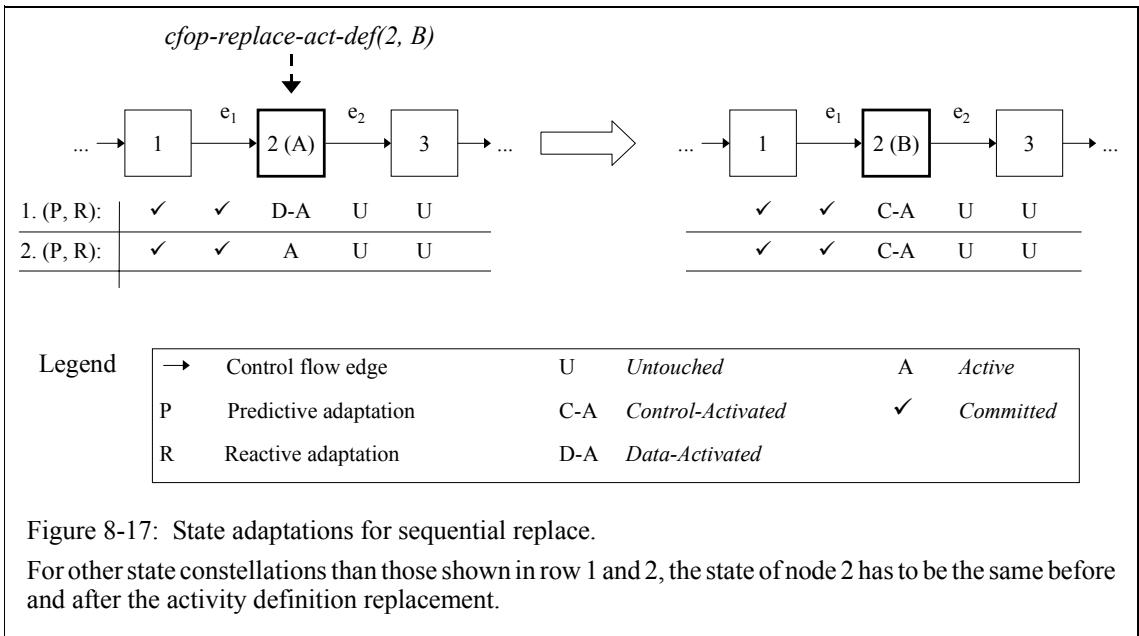
For node postponement, AGENTWORK provides the control flow operator

cfop-postpone-node(*n*: Integer, *d*: Distance).

The first input parameter *n* identifies the node to be postponed, and the second one *d* the temporal distance $d > 0$ by which *n* shall be postponed. This operator is invoked when a node *n* is affected by a *postpone*(*A*,*d*,*C*) or *postpone-activities-of*(*R*,*d*) control action.

8.2.7.1 Effect on Control Flow without Loops, Synchronization and Waiting Edges

As described in Chapter 7, the semantics of *postpone*(*A*,*d*,*C*) and *postpone-activities-of*(*R*,*d*) is that



the temporal distance d between the old and the new relative position of a postponed activity execution may not be changed by subsequent adaptations. Therefore, *cfop-postpone-node* does not try to postpone n by reordering the sequence to which n belongs, as the nodes between the old and new position may be subject of further adaptations. Rather, it provides different mechanisms to postpone the execution of n . Analogously to sequential and parallel add resp. replace, we distinguish between *sequential* and *parallel* postpone.

Sequential postpone: The straightforward way to postpone the execution of n is to simply add a waiting condition $\min = \max = d$ between n and its successor node, and to mark the respective edge as *Non-Adaptable*. (Figure 8-18). However, the disadvantage of sequential postpone is that it may postpone successor nodes of n as well (e.g., node 3 in Figure 8-18). Thus, goal 3 (*Minimization of Execution Delay*) would be violated. Therefore, *cfop-postpone-node* does not use sequential postpone as default strategy, but parallel postpone which is described now.

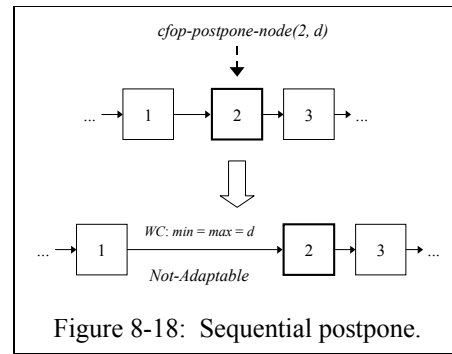


Figure 8-18: Sequential postpone.

Parallel postpone: To avoid execution delays of other nodes than the node to be postponed, *cfop-postpone-node* by default tries to source n out to an own path within a new AND-SPLIT/AND-JOIN block (Figure 8-19). For this, *cfop-postpone-node* invokes *cfop-gen-empty-parallel-path* with the following parametrization:

- Analogously to *cfop-replace-act-def*, the first parameter of *cfop-gen-empty-parallel-path* is set to the successor node of the node n by which *cfop-postpone-node* has been invoked.
- The second parameter is set to $d + \text{dur-av}(A)$ with $A = \text{NAM}(n)$ (i.e., A is the activity definition assigned to n). This is necessary as the execution duration of the path to which n shall be sourced out consists first of the postponement duration (i.e., d) and second of the assumed execution duration of n (i.e., $\text{dur-av}(A)$).

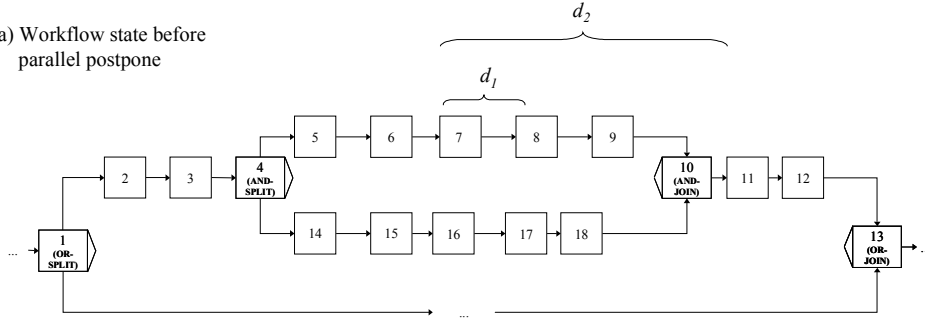
If *cfop-gen-empty-parallel-path* is able to generate a new AND-SPLIT/AND-JOIN block, node n is moved to the new parallel path. In particular, *cfop-drop-node* is invoked to drop node n from its original path. For example, in Figure 8-19 b) node 7 is dropped from its original sequence and added to the new path parallel to nodes 8 and 9. Furthermore, a waiting condition with duration $\min = \max = d_l$ is assigned to the edge between the new AND-SPLIT node 19 and the moved node 7. Finally, the edge with the waiting condition is marked as *Non-Adaptable*. By such a parallel postpone, it is achieved that the execution of node 7 is delayed by duration d_l without delaying the former successor nodes of node 7 (e.g., nodes 8 and 9).

However, a problem remains if *cfop-gen-empty-parallel-path* has not been able to insert the new AND-SPLIT node directly at the old position of n but somewhere before (as shown in Figure 8-19 c). Then it does not make much sense to assign a waiting condition of duration d to the edge between the new AND-SPLIT node and n (e.g., between node 19 and node 7 in Figure 8-19 c) as

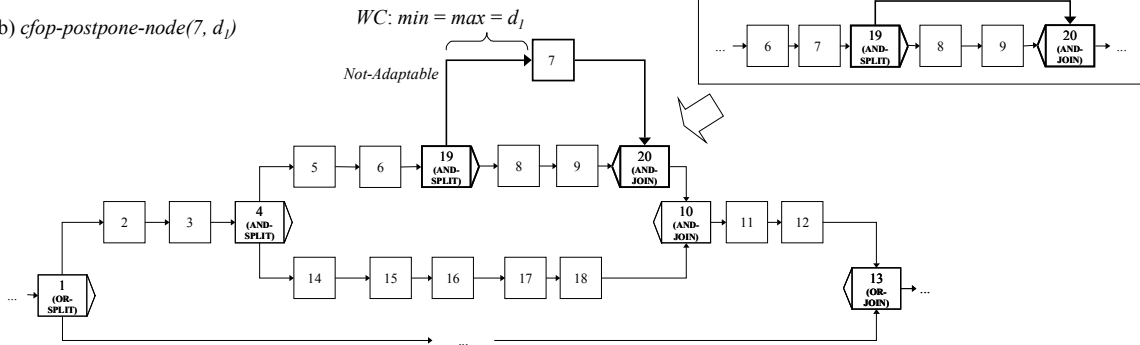
Control Flow Operators

$cfop_postpone_node(7, d_i)$ ($i = 1, 2$)

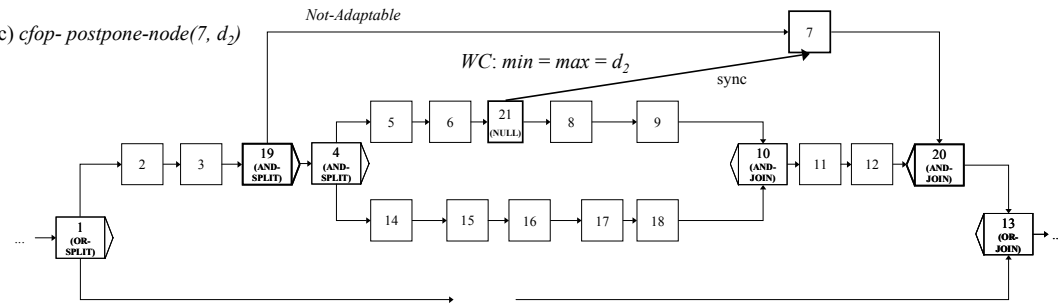
a) Workflow state before parallel postpone



b) $cfop_postpone_node(7, d_1)$



c) $cfop_postpone_node(7, d_2)$



Legend WC: Waiting condition

Figure 8-19: Parallel postpone.

this would not anymore be a postponement w.r.t. the old position of n . It also does not make much sense to add the duration of the path between the new AND-SPLIT node and the old position of n to the waiting condition, e.g., to add the duration of path $4 \rightarrow 5 \rightarrow 6$, as this path may be changed by further adaptations and thus may change its duration. Therefore, *cfop-postpone-node* inserts a NULL node (see 8.1.3.2) at the former position of n , synchronizes n with this NULL node by inserting the synchronization edge

$$((NULL \text{ node}, Committed), (n, Control-Activated)) \quad (ii)$$

to the workflow, and assigns a waiting condition with $min = max = d$ to this synchronization edge to achieve the node postponement. For example, in Figure 8-19 c) the NULL node 21 is inserted between node 6 and 8 (i.e., the *old* position of node 7), and node 7 is synchronized with it by an synchronization edge of form (ii) to which a waiting condition of duration d_2 is assigned.

If node n cannot be sourced out to a parallel path as it for example belongs to a logical sequence that would be violated, or if *cfop-gen-empty-parallel-path* is not able to generate a new AND-SPLIT/AND-JOIN block (e.g., returns FALSE), *cfop-postpone-node* performs a sequential postponement.

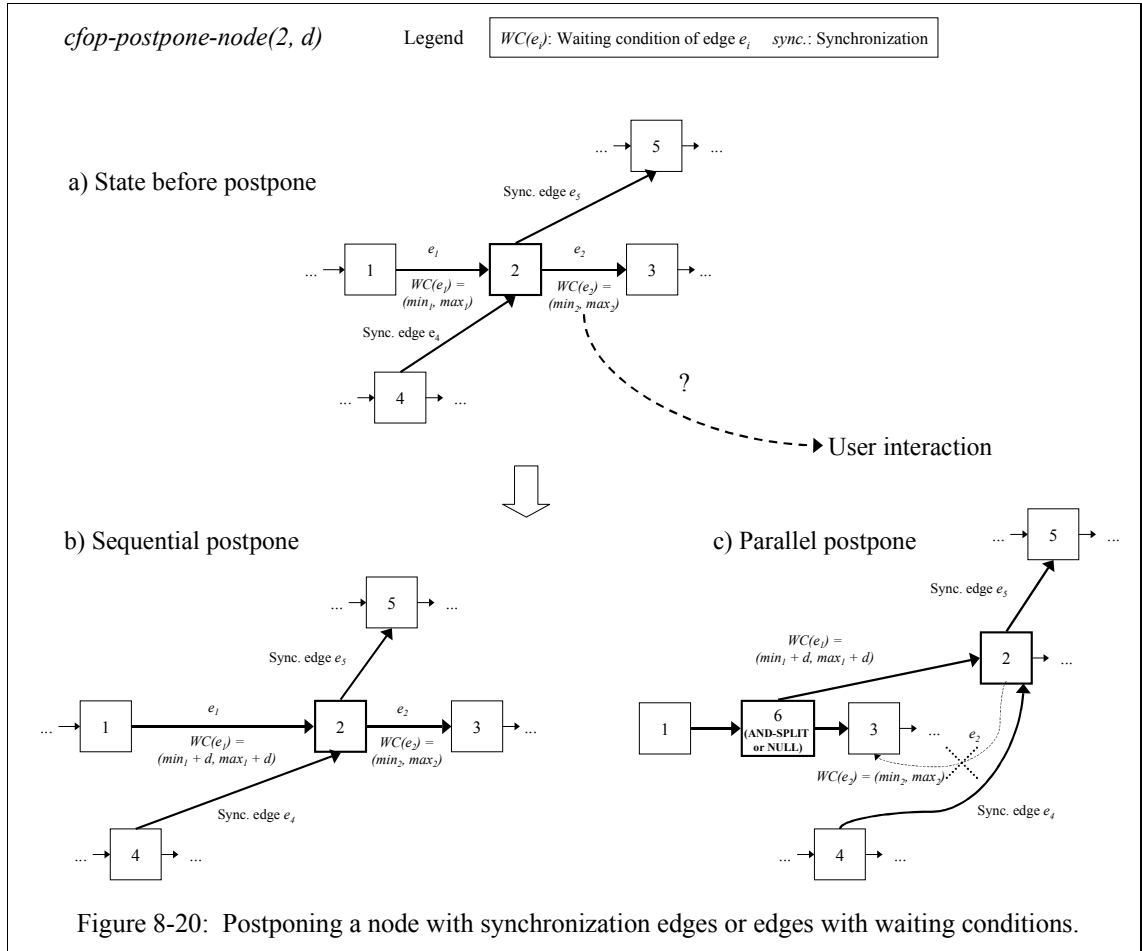
8.2.7.2 Handling of Loops

Concerning the handling of loops, the statements made for *cfop-replace-act-def* (8.2.6.2) hold analogously for *cfop-postpone-node*. This is because the only structural difference between the operators *cfop-replace-act-def* and *cfop-postpone-node* is that the first changes an activity definition of an existing node, while the latter adds a waiting condition to an incoming control flow or synchronization edge of an existing node.

8.2.7.3 Handling of Synchronization and Waiting Edges

If a node n which has to be postponed is the source or target of synchronization edges or edges with waiting conditions, AGENTWORK principally assumes that these synchronization edges or edges with waiting conditions should be maintained. This is because postponing n means that the activity semantics of n is kept (in contrast for example to an activity definition replacement). Thus, temporal dependencies of n to other nodes are kept, as shown in Figure 8-20. In particular, if n is postponed sequentially *and* is the target of an edge with waiting condition (Figure 8-20 b), the postponement duration d has to be added to the already existing waiting condition.

However, an exception is that n is the source of an edge with waiting condition (e.g., node 2 in Figure 8-20 a) and is postponed by *parallel* postpone (Figure 8-20 c): On one side, maintaining the edge with the waiting condition – e.g., assigning the waiting condition $WC(e_j) = (min_j, max_j)$ to the edge (2,3) in Figure 8-20 c) – would mean that the main motivation for parallel postpone would be counteracted, namely the avoidance of execution delays w.r.t. former successor nodes of n . On the other side, assigning the waiting condition to the edge between the inserted AND-SPLIT or NULL node and the former successor node of n (e.g., between nodes 6 and 3 in Figure 8-20 c) often will not make much sense, as the temporal constraint expressed by the waiting condition typically



depends on the activity definitions of the connected nodes. Thus, if one of the both nodes does not “carry” the activity definition for which the waiting condition has been defined (e.g., node 6 in Figure 8-20 c), this waiting condition often will become obsolete. Summarizing, we can state that an automated handling of this constellation of parallel postpone is difficult, so that the user has to be requested: If he wants to maintain the waiting condition between n and its former successor node, a sequential postpone is performed instead of a parallel postpone (as the latter then does not make any sense). Alternatively, the user can either decide to drop the waiting condition entirely, or to assign it to the edge between the AND-SPLIT resp. NULL node and the former successor node of n (e.g., between node 6 and 3 in Figure 8-20 c).

Another conflict situation that may occur is that n is the target or source of a synchronization edge e (e.g., of edge e_1 resp. e_2 in Figure 8-21), and that e also contains a waiting condition. Then, independently from whether a sequential or parallel postpone is performed for n , the following two sit-

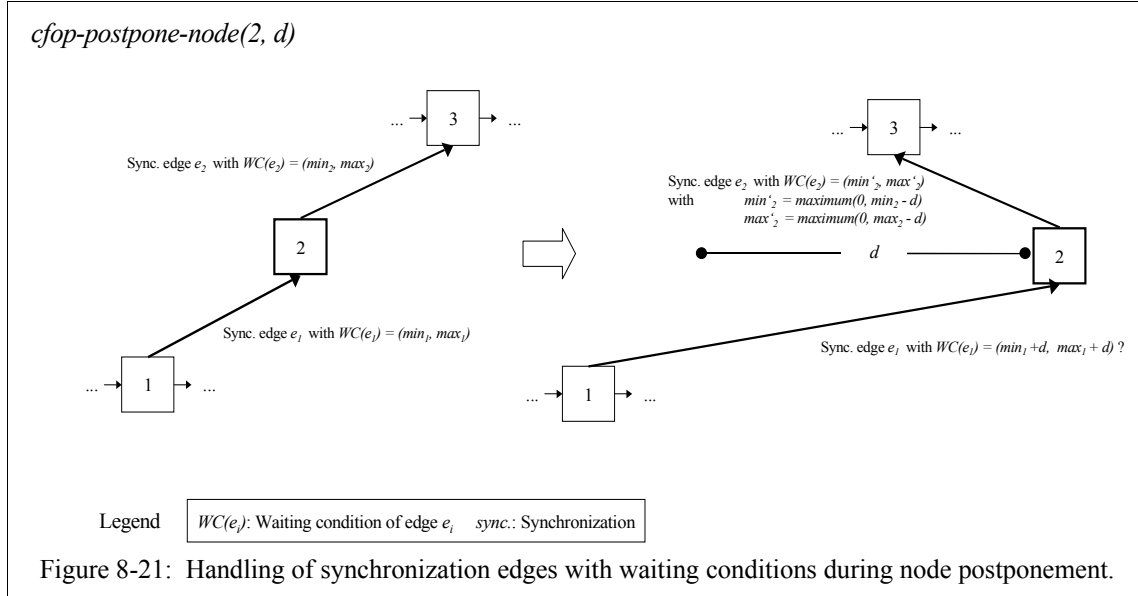


Figure 8-21: Handling of synchronization edges with waiting conditions during node postponement.

uations may occur:

Situation 1: If n is the target of $e = ((m, s), (n, t))$ (e.g., $e = e_1$ in Figure 8-21), the max entry of $WC(e)$ may be not satisfiable anymore due to the later execution of n . Recall from 5.4.2.1 that the max entry of $WC(e) = (min, max)$ has the semantics that a necessary precondition to set n to state t is that since m has been set to state s at most the time specified by max must have elapsed. At first glance, an appropriate way to cope with this seems to be to add d to the min and max entry of the waiting condition (as shown in Figure 8-21), as this reestablishes the temporal consistency of the workflow. However, this may not always be appropriate. For example, if n and m support drug applications with a strict waiting condition between them because of the metabolism behavior of the drugs, an automated adding of d would be inappropriate or even fatal from the medical point of view. As this cannot be determined automatically, the only way to cope with this is to request the user who can decide to add d to the min and max entry or to adjust min and max differently, to remove the synchronization edge, or to reject the node postponement *at all* if he weights the synchronization edge stronger than the node postponement.

Situation 2: If n is the source of $e = ((n, s), (m, t))$ (e.g., $e = e_2$ in Figure 8-21), the point in time when m can be set to state t may be significantly delayed after n has been postponed. Similar to situation 1, it may appropriate to adjust the min and max entry of $WC(e)$ as

$$\begin{aligned} min &= maximum(0, min - d), \\ max &= maximum(0, max - d), \end{aligned}$$

to remove the synchronization edge or to reject the node postponement *at all*. As this cannot be

determined automatically, the user has to decide.

What remains is the question how situation 1 and situation 2 can be *detected* at all. This can be done in a straightforward manner by the estimation algorithms introduced in Chapter 6 by comparing the execution durations of the workflow before and after the node postponement. Thus, we omit details here.

8.2.7.4 Side-Effects on Data Flow

As *cfop-postpone-node* only moves a node n without changing its activity definition, this operator has no direct effect on *existing* data flow edges. However, if the postponed node n provides output objects that are needed by other nodes or for condition evaluations, the postponement of n may imply a substantial data flow-induced delay of these dependent nodes because internal data flow edges between n and these nodes or conditions have the effect of synchronization edges (which has been already a problem for workflow estimation as described in Chapter 6). Thus, it may be necessary to remove such delaying data flow edges and to insert an alternative data flow edge which does not cause such an execution delay. This will be described in 8.3.

8.2.7.5 State Adaptations

The state adaptations that have to be performed by *cfop-replace-act-def* (8.2.6.5) hold analogously for *cfop-postpone-node*. The only difference is that if the node to be postponed already is in state *Data-Activated*, this state can be kept also for the postponed node (see row 1 in Figure 8-17), given that no time constraints of its input object patterns (see 5.2.2) are violated by the postponement.

8.3 Data Flow Adaptation

We now describe how AGENTWORK adapts the data flow after a control flow adaptation. For this, we first describe the *conditions* under which such a data flow adaptation becomes necessary (8.3.1). Second, we describe the principal strategies of data flow adaptation (8.3.2). Third, we describe an operator that adapts the data flow (8.3.3).

8.3.1 Conditions for Data Flow Adaptation

The different effects the control flow operators of Section 8.2 might have on a workflow's data flow have already been sketched informally in the resp. section *Side-Effects on Data Flow*. We now give a precise description under which conditions a data flow adaptation becomes necessary. Please recall from 5.3.6 (*Data Flow Definitions*), that we assume workflow-wide *unique* names for all input, output, and communication objects.

A control flow adaptation requires a data flow adaptation if it exists an activity node, a branching condition, or a COMM-OUT node x , for which one of the following two conditions holds:

- a) *Incomplete Input*: It exists $p \in \text{input}_x$ (if x is an activity node or branching condition) resp. $p \in \text{comm-objs}_x$ (if x is a COMM-OUT node) with:

There is no data flow edge $(o, p) \in \text{Internal-Data-Flow} \cup \text{External-Data-Flow}$.

- b) *Data Flow-Induced Execution Delay*: It exists $p \in \text{input}_x$ resp. comm-objs_x and a data flow edge $(o, p) \in \text{Internal-Data-Flow} \cup \text{External-Data-Flow}$ with

$$\text{entry-of-edge-state}((o, p), \text{Committed}) \leq \text{entry-of-node-state}(x, \text{Control-Activated}) \quad (\text{iii})$$

before the control flow adaptation but

$$\text{entry-of-edge-state}((o, p), \text{Committed}) > \text{entry-of-node-state}(x, \text{Control-Activated}) \quad (\text{iv})$$

after the control flow adaptation. According to 6.1.3.1 (*Significance of Durations*), the “>” operator in (iv) has to be understood in the sense of “significantly” later.

Condition a) means that at least one object needed by an activity node, a branching condition, or a COMM-OUT node is neither provided by the internal nor by the external data flow. Thus, data flow constraint 1 (input completeness) is violated. Condition a) may become true when the control flow operators *cfop-drop-node*, *cfop-add-node*, *cfop-add-node-loop*, or *cfop-replace-act-def* are applied to a workflow.

Concerning condition b), (iv) means that after an adaptation some data flow edge (o, p) can only perform its object mapping from o to p after the point in time at which x has been set to state *Control-Activated* (and thus needs p). In contrast to this, before the adaptation (o, p) has been able to map its data before the point in time at which x has been set to state *Control-Activated* (iii). Condition b) may become true when the control flow operators *cfop-postpone-node*, *cfop-add-node*, or *cfop-replace-act-def* are applied to a workflow as they all may induce execution delays of the affected node itself or of its successor nodes⁷.

Independently from whether condition a) or b) holds, we have the situation that for an activity node, a branching condition, or a COMM-OUT node x an input resp. communication object p exists for which no appropriate data flow edge (o, p) exists to fill p . It's the task of the operator introduced in 8.3.3 to generate such an edge (o, p) and to suggest it to the user.

8.3.2 Data Flow Adaptation Strategies

Before we describe the concrete structural data flow adaptations in 8.3.3, we first sketch the two principal strategies that can be identified for data flow adaptation, namely *reactive* and *predictive* data flow adaptation (analogously to reactive or predictive adaptation of a workflow's control flow). To describe these two strategies, we again assume that x is an activity node, a branching condition, or a COMM-OUT node, i.e., a workflow element needing data objects for its execution.

7. Note that an *add-repetitively* control action cannot induce an execution delay (if resource availability holds), as it is always translated into a loop parallel to the rest of the workflow which cannot delay other node or edge executions (except the END node).

8.3.2.1 Reactive Data Flow Adaptation

This strategy means that conditions a) and b) are checked directly before x needs its data objects, i.e., when x is set to state *Control-Activated*. If one of these two conditions holds, the data flow is adapted. Obviously, this data flow adaptation strategy can be combined both with reactive and predictive control flow adaptation (the latter case means that the control flow is handled predictively, but the necessary data flow adaptations are performed “on demand”).

The reactive data flow adaptation strategy has the advantage that the need of temporal estimations can be reduced: On one side less temporal estimations may be necessary to evaluate the inequations (iii) and (iv), as the point in time when x is set to state *Control-Activated* – i.e., $\text{entry-of-node-state}(x, \text{Control-Activated})$ – is definitely known. Note that at least for the left side of (iv) a temporal estimation may still be necessary as this is a point in time of the *future* w.r.t. $\text{entry-of-node-state}(x, \text{Control-Activated})$.⁸

Analogously to reactive control flow adaptation, the disadvantage of reactive data flow adaptation is that it may be too late. For example, if a new therapeutic node is added to a medical workflow, and if this new node needs an x-ray finding as input, it may turn out during reactive data flow adaptation that such an x-ray finding is not available at all for the respective patient, so that an x-ray-examination first has to be executed. This costs time so that the execution of the therapeutic node is delayed. It could be argued that at least for the combination of reactive data flow adaptation with *predictive* control flow adaptation, such required data-producing nodes (e.g., the x-ray examination node) could be added automatically to the workflow by predictive control flow adaptation (by extending the respective failure rules) so that the sketched problem cannot occur. However, this would require that *all* activity types of an organization are represented electronically within the workflow system. This must be viewed as unrealistic especially w.r.t. the first phases of bringing a workflow system into practice.

8.3.2.2 Predictive Data Flow Adaptation

This strategy means that conditions a) and b) are checked for x directly after the control flow adaptation. If one of these two conditions holds, the data flow is adapted predictively, i.e., while x still is in state *Untouched*. At first glance, this data flow adaptation strategy seems to be usable only after predictive control flow adaptation. However, note that also for a reactive control flow adaptation such as reactively dropping a node n , a successor node or conditional successor edge x of n that is still in state *Untouched* may meet condition a) or b). Thus, it is principally possible to predictively adapt the data flow though the control flow is adapted reactively.

Concerning advantages and disadvantages of predictive data flow adaptation, the situation is diametrical to reactive data flow adaptation: The advantage is that data flow adaptations that require new data-producing activity nodes (such as the x-ray examination node of the example sketched in

8. Again, note that the usage of reactive control flow adaptation does not principally exclude that path durations are estimated (as the missing of duration information or the existence of unresolvable conditions are only two of several possible reasons for using reactive adaptation).

8.3.2.1) are done in time. The disadvantage is that to evaluate the inequations (iii) and (iv) for x , much more temporal estimations are needed as for reactive data flow adaptation. This is because the right sides of (iii) and (iv) are points in time of the *future*, as x is still in state *Untouched*. Furthermore, for the data flow adaptation itself temporal estimations will be needed.

8.3.2.3 Strategy Selection for Data Flow Adaptation

The question remains when to use which data flow adaptation strategy. For AGENTWORK the answer is simple: Predictive data flow adaptation is used whenever possible, i.e., when the temporal estimations needed are possible. This is because the disadvantage of reactive data flow adaptation, i.e., that necessary data flow adaptation may be performed too late, is weighted stronger than the disadvantage of predictive data flow adaptation, i.e., the increased number of necessary temporal estimations.

8.3.3 Operator for Adding Data Flow Edges

The generation and suggestion of an appropriate data flow edge needed to overcome the situations described by conditions a) resp. b) in Section 8.3.1 is performed by the data flow operator

dfop-gen-data-flow-edge(x : Integer, p : Time-Constr-Named-Obj-Patt).⁹

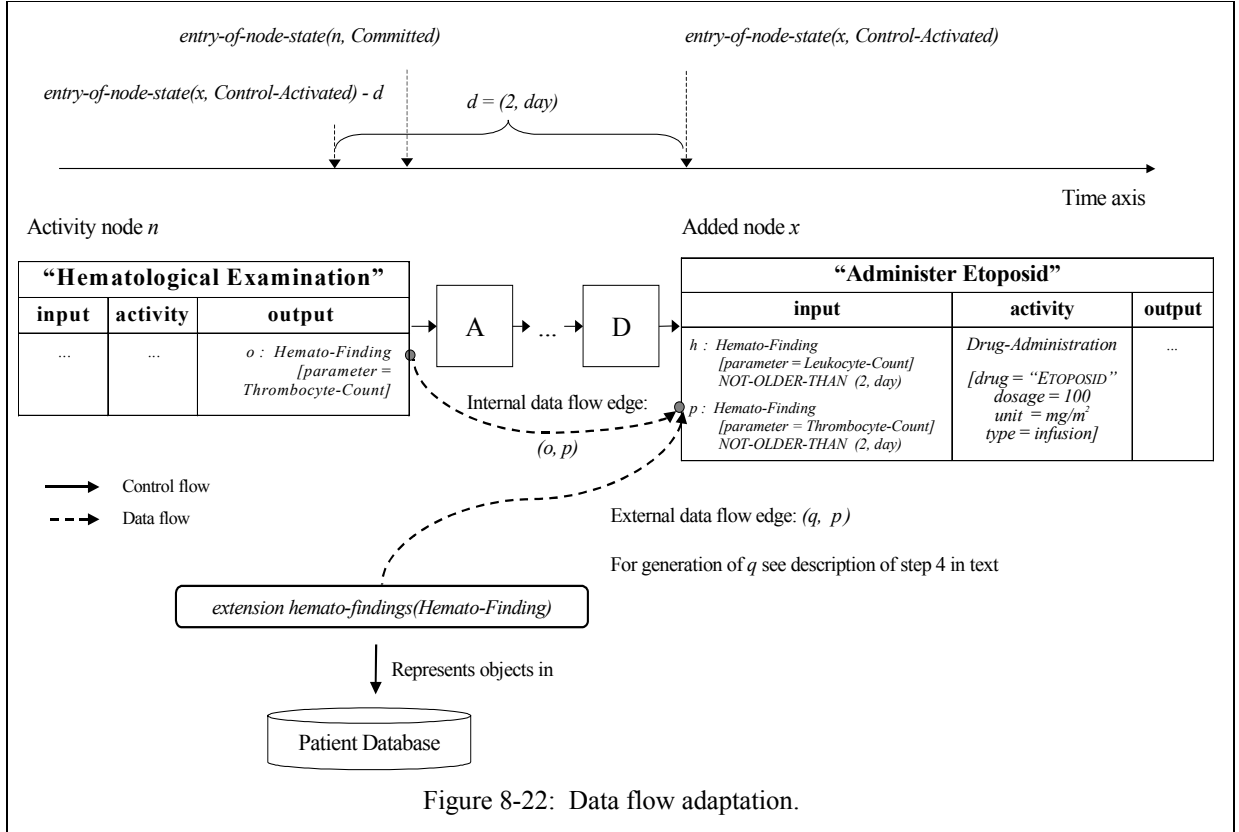
This operator takes as its first input parameter an identifier x of a node or a branching condition for which an appropriate data flow edge has to be generated. The second input parameter p is the input object needed by x , i.e., $p \in \text{input}_x$ resp. comm-objs_x (more precisely, p is the time-constrained named object pattern specifying the object needed as input).

For a compact presentation of *dfop-gen-data-flow-edge*, we assume that x is an *activity node* for which a data flow edge has to be generated, and that the workflow to which it belongs does not contain COMM-IN nodes (which could serve as potential data sources for x as well). For x being a communication node or an edge with branching condition, and for workflows containing COMM-IN as potential data sources for x , data flow adaptation is handled analogously.

The principal way *dfop-gen-data-flow-edge* works is as follows: First, *dfop-gen-data-flow-edge* explores the “relevant” temporal neighborhood of x and checks whether there is any output object o of an activity node n that matches the pattern and time constraint that p has to fulfill (Steps 1 and 2). If this is the case, an *internal* data flow edge is generated that maps o to p (Step 3). If the local temporal neighborhood does not provide such an object o , *dfop-gen-data-flow-edge* generates an F-Logic query q to an object extensions and constructs an *external* data flow edge that maps the object retrieved by q to p (Step 4). Finally, the generated internal or external data flow edge is presented to the user, which may confirm, reject, or adapt it manually (Step 5).

We explain these steps now in more detail. To illustrate this, we assume that due to the application of *cfop-add-node* an activity node x supporting an ETOPOSID administration has been added to a

9. *dfop* stands for “data flow operator”



workflow (Figure 8-22) and that for x an input object p is required that fulfills the pattern

$$p: \text{Hemato-Finding} [\text{parameter} = \text{Thrombocyte-Count}] \text{ NOT-OLDER-THAN } (2, \text{day}) \quad (v)$$

i.e., a *Hemato-Finding* object named p is needed which stores a thrombocyte count and which may not be older than 2 days when x is executed.

Step 1: Determination of “Relevant” Temporal Neighborhood

The relevant local neighborhood of x is the set of those activity nodes that are set to state *Committed*

- not significantly earlier than the point in time when x is set to state *Control-Activated* minus the distance specified by the *NOT-OLDER-THAN* constraint of p , and
- not significantly later than the point in time when x is set to state *Control-Activated*.

Formally, this is the set

$N_{x,d} = \{n \mid n \text{ activity node with:}$

$$\begin{aligned} & \text{entry-of-node-state}(n, \text{Committed}) \in \\ & [\text{entry-of-node-state}(x, \text{Control-Activated}) - d, \\ & \text{entry-of-node-state}(x, \text{Control-Activated})], \end{aligned} \quad (\text{vi})$$

where d denotes the temporal distance specified in the *NOT-OLDER-THAN* constraint of p . In the example of Figure 8-22, the node n supporting the hematological examination belongs to $N_{x,d}$.

The definition of $N_{x,d}$ makes sense as in state *Committed* a node n can provide all its output objects, and as in state *Control-Activated* x needs the missing object p . Note that it is not required that n is a predecessor node of x , as also nodes executed parallel to x are candidates for providing output objects for x .

If x is still in state *Untouched* and depending on the workflow constellation around x , temporal estimations as described in Chapter 6 (*Workflow Duration Estimation*) may be necessary to determine $N_{x,d}$. Of course, if the data flow adaptation is performed after a predictive control flow adaptation, some of the necessary estimations may already have been performed when determining the workflow part P_{VT} corresponding to a valid time VT of a control action, and thus can be reused.

Step 2: Object Pattern Matching

After the set $N_{x,d}$ has been determined, *dfop-gen-data-flow-edge* checks whether any member of this set provides an appropriate output object w.r.t. p . Formally, this means that for every $n \in N_{x,d}$ and every $o \in \text{output}_n$ the following conditions have to be checked:

- a) Concerning the cases $\text{case}(n)$ and $\text{case}(x)$ (5.4.3.5) for which n resp. x are executed, it holds:

$$\text{case}(n) \text{ and } \text{case}(x) \text{ are already known, and} \quad (\text{vii})$$

$$\text{case}(n) = \text{case}(x). \quad (\text{viii})$$

This means that both nodes n and x are executed for the same case and thus that any output object provided by n is for (i.e., represents data concerning) exactly that *Case* instance for which p is needed. For the example in Figure 8-22, let us assume that nodes n and x are executed both for the same patient, so that both the output object o of n and p represent data of the same patient.

- b) It holds

$$o \text{ matches } p \text{ according to 4.2.1.7.} \quad (\text{ix})$$

For example, in Figure 8-22 the output object o meets the pattern p , as o is a *Hemato-Finding* object representing a thrombocyte count, too.

Note that for *reactive* data flow adaptation (vii) is always fulfilled. This is first because for this strategy an affected node x is handled not before it has been set to state *Control-Activated* (7.4.3.1) and as according to our execution model of Chapter 5 a case must have been assigned to x so far (as otherwise x could not be executed). Second, due to the definition in (vi) a node n in $N_{x,d}$ has committed not later than x is set to state *Control-Activated*, so that for reactive data flow adaptation the case assigned to n is already known, too. In contrast to this, (vii) must not always be fulfilled if *predictive* data flow adaptation has been selected. For example, $case(n)$ may be unknown if the case is assigned dynamically to n and if n has not been executed at the moment of the data flow adaptation. As a consequence, such a node n with yet unknown case cannot qualify as source for an internal data flow edge.

Step 3: Generation of Internal Data Flow

If there exists an $n \in N_{x,d}$ and an $o \in output_n$ that fulfills (vii) – (ix), the internal data flow edge (o, p) is inserted into the set of internal data flow edges *Internal-Data-Flow*. For example, in Figure 8-22 the edge (o, p) is inserted between n and x to transfer the thrombocyte count information between these two nodes.

If there is more than one pair (n, o) fulfilling the required conditions, the most “current” n is selected, i.e., that n with the smallest temporal distance w.r.t. x :

$$\begin{aligned} & entry-of-node-state(n, Committed) - entry-of-node-state(x, Control-Activated) = \\ & \min \{ entry-of-node-state(n', Committed) - \\ & \quad entry-of-node-state(x, Control-Activated), n' \in N_{x,d} \} \end{aligned} \quad (x)$$

If at least two n fulfill equation (x) (i.e., have the same minimal temporal distance to x), then one n is selected at random. If the selected n has two objects o, o' fulfilling the pattern condition (ix), then one is selected at random, too.

Step 4: Generation of External Data Flow

If $N_{x,d}$ cannot be determined at all (as the necessary estimations are not possible), or if there is no $n \in N_{x,d}$ that fulfills (vii) – (ix), *dfop-gen-data-flow-edge* generates an F-Logic query to retrieve an object o meeting the pattern and time constraint specified by p . This query is generated on the base of the template shown in Table 8-2 (left and middle column). In this table, the function

$$insertion-or-last-update(o: Object): T \quad (xii)$$

used in the *time-constr* row of Table 8-2 returns the point in time when o was inserted or last updated into its extension. The condition (xi) in this row corresponds to condition (vi) needed for the definition of $N_{x,d}$ in step 1 (i.e., instead of the point in time when o is provided as output object of an activity node, we have to take the point in time when o is inserted or last updated in its extension).

Template Part	Meaning	Example (Figure 8-22)
?-	Query operator	?-
o IN	Object to retrieve	o IN
extension	The extension to be queried (Note that we assume a 1:1 relationship between F-Logic classes and extensions; see 4.2.1.8).	hemato-findings
AND		AND
case-filter	The case to which o has to refer, i.e., $case(x)$	$o.of = case(x)$ ($o.of$ refers to the <i>Case</i> instance to which o belongs to; see 3.2.1, 4.2.1.2).
AND		AND
obj-pattern	The pattern that o has to fulfill, i.e., p (without NOT-OLDER-THAN constraint)	$o.parameter = Thrombocyte-Count$
AND		AND
time-constr	The time constraint o has to fulfill concerning its currentness, i.e., $insertion-or-last-update(o) \in$ $[entry-of-node-state(x, Control-Activated) - d, entry-of-node-state(x, Control-Activated)]$ (xi)	$insertion-or-last-update(o) \in$ $[entry-of-node-state(x, Control-Activated) - d, entry-of-node-state(x, Control-Activated)]$

Table 8-2: Template and example for F-Logic query generation.

For the workflow in Figure 8-22, the query q shown in Table 8-2 (right column) has been generated. This query q is then part of an external data flow edge (q, p) which is inserted into the set of external data flow edges *External-Data-Flow* and executed when x is executed.

Step 5: User Request

It cannot be guaranteed in general that generated data flow edges really do provide the right data. One reason for this may be underspecified object patterns. For example, imagine that a node n needs some *Radiodiagnostic-Activity* object as input, but that the focus of this activity (e.g., lung, head etc.) has not been specified via the *focus* attribute of *Radiodiagnostic-Activity* class. Then, according to steps 1-3, AGENTWORK would search for *Radiodiagnostic-Activity* output objects in the appropriate temporal neighborhood without considering the *focus* attribute. If we then assume that two *Radiodiagnostic-Activity* output objects o_1, o_2 are in the relevant temporal neighborhood, the more current *Radiodiagnostic-Activity* output object would be selected (e.g., o_1). However, this

may be exactly the wrong one, if n would require a *Radiodiagnostic-Activity* object with $focus = \text{“Lung”}$, but if it would be $o_1.focus = \text{“Head”}$ and $o_2.focus = \text{“Lung”}$.

As such constellations caused by underspecified input patterns cannot be excluded, the last step of data flow generation has to consist of presenting the generated data flow edge to the user and to request whether it is appropriate w.r.t. the workflow semantics. The user can confirm, adapt, or reject it. In the latter case, the system then suggests other data flow edges according to steps 1-4. If there are no other data flow edges meeting the required conditions, or if the user rejects all of them, he can enter the needed data manually (5.4.3.2).

Recall from the discussion in 5.3.6.4, that due to conditional branching even a generated *and* confirmed (internal) data flow edge e does not necessarily mean the needed data are present at execution time (as the source node of e may not have been executed at all). In this case, the user is also requested to type in the missing data manually.

Summarizing, it can be stated that though required data flow edges cannot be generated in an entirely automated manner, the described approach significantly reduces the effort of data flow adaptation due to its comprehensive suggestions based on temporal estimation and pattern matching.

8.4 Summary and Discussion

In this chapter we have described the adaptation operators that translate the local control actions of Chapter 7 into structural workflow adaptations, i.e., into adaptations of a workflow’s node and edge set. In particular, we have described the *control flow* adaptation operators which translate local control actions into structural control flow adaptations. The main characteristic of these control flow operators has been that whenever possible they use the technique of generating new parallel paths in order to add new nodes or to outsource existing nodes to such new parallel paths (parallel add, parallel replace, and parallel postpone). This is done first to minimize execution delays caused by workflow adaptations and thus to reduce push-out effects, and second to avoid splitting up logical sequences. Concerning the minimization of execution delays, it is important to note that the control flow operators achieve this independently from the question whether predictive *or* reactive (control flow) adaptation is used, though reactive adaptation has no notion of the workflow part corresponding to the valid time of a control action.

As a control flow adaptation may require that the data flow is adapted as well, we have also introduced mechanisms and in particular a data flow adaptation operator for the generation of required data flow edges after a control flow adaptation. In particular, we have seen that analogously to reactive and predictive control flow adaptation the data flow can be adapted reactively or predictively too, and that both reactive and predictive control flow adaptation can be combined with reactive or predictive data flow adaptation.

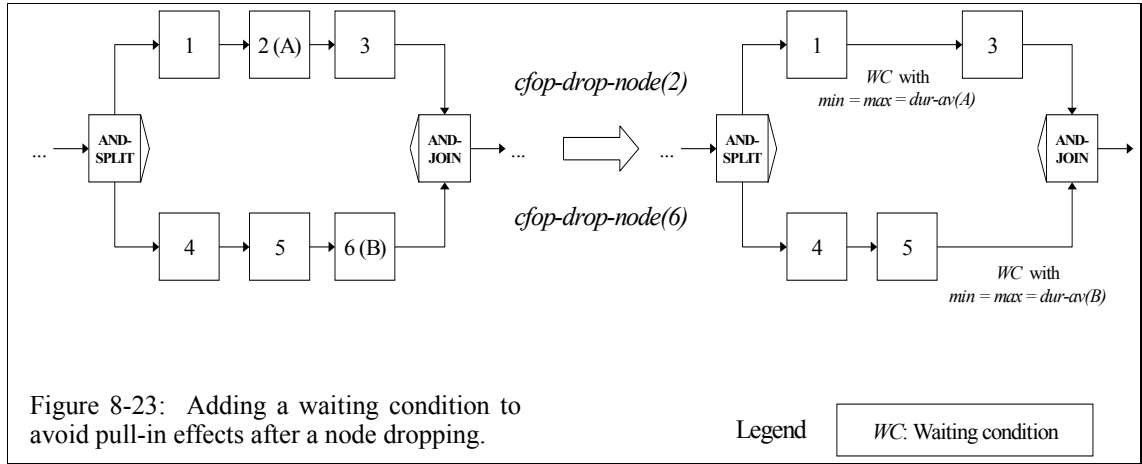
Several points of criticism can be made concerning the introduced adaptation operators and in particular the “parallel” versions of the control flow operators:

1. The complexity of structural workflow adaptation and in particular of parallel add, parallel replace, and parallel postpone is high and therefore hard to manage. This holds especially if multiple control actions affect a workflow simultaneously. In particular, if the affected workflow part contains loops the adaptations may become very complex as then further conditional branchings may have to be added to the workflow. As then quite a lot of user interactions may be necessary, the user may be overcharged. This will decrease the acceptance of the system.
2. An AND-SPLIT/AND-JOIN block generated by parallel add, parallel replace, or parallel postpone may be affected by further adaptations. Thus, estimations that have been performed to satisfy the temporal optimization Criterion 8.1 may become invalid so that the intended avoidance of execution delays and push-out effects is not achieved anymore.
3. The generation of AND-SPLIT/AND-JOIN blocks and the achieved avoidance of execution delays is justified by the assumption of resource availability (8.1.2) as this assumption allows to execute the paths of an AND-SPLIT/AND-JOIN block in parallel *in-fact* (without any implicit serialization). However, if this assumption does not hold for a particular workflow, the complexity that parallel add, parallel replace, and parallel postpone bring into this workflow is not justified anymore.

The main counter-argument concerning points 1 and 2 of criticism is that due to assumption 1 (*Limited Number of Simultaneously Triggered Control Actions*) introduced in 8.1.2, we do not expect too many control actions (i.e., more than 2 or 3) affecting one workflow simultaneously on the average. Thus, it can be assumed that the complexity remains manageable as subsequent adaptations do not occur too often. As in many applications only “expert” staff members (e.g., physicians, not nurses) should have the competence to adapt workflows, it can be assumed that these experts are able to deal with the complexity of required user interactions. If it turns out for some applications that the complexity of user interactions overcharges even these “expert” users, there would still be the possibility to program enhanced user interfaces that allow operations on a high-level of abstraction and hide as many as possible syntactical details.

Furthermore, the complexity of parallel add, parallel replace, and parallel postpone is justified due to the avoidance of execution delays. Concerning point 3 it can be argued that even when this execution delay cannot be avoided as the resource availability assumption does not hold for a particular workflow, parallel add, parallel replace, and parallel postpone first *do not produce more execution delay* than their sequential counterparts, and second *on the average produce less execution delay* than their sequential counterparts.

Nevertheless, it must be stated that though the control flow operators of this chapter minimize execution delays and thus push-out effects by generating new parallel paths, they of course cannot always avoid execution delays and thus push-out effects, namely when the sequential versions (i.e., sequential add, sequential replace and sequential postpone) have to be performed. This has the consequence that if these control flow operators are invoked during *reactive* control flow adaptation, push-out effects may occur but are not even noticed and thus cannot be controlled at all (e.g., by requesting the user to decide how to cope such a push-out effect). This of course is not really a



point of criticism affecting the control flow operators themselves but one affecting reactive control flow adaptation, as this strategy has no notion of the workflow part corresponding to the valid time of a control action, and thus cannot control all push-out effects.

Furthermore, one may wonder that on one side the control flow operators provide mechanisms to minimize execution delays and thus *push-out* effects but that on the other side no mechanisms are provided by them to minimize *pull-in* effects. Concerning this problem, one could argue an appropriate mechanism to minimize pull-in effects could be to add waiting conditions to an adapted workflow whenever nodes are dropped or whenever an activity definition is replaced by one having a shorter execution duration (e.g., to assign in Figure 8-23 a waiting condition with the duration of the activity definition of the dropped node 2 to the edge (1,3) after the adaptation). Then, successor nodes cannot be executed earlier and thus cannot be pulled into a valid time interval of a control action. In addition to this, the waiting time could be viewed as a sort of a free temporal slot into which further nodes could be inserted during subsequent adaptations to minimize push-out effects. By this mechanism of adding waiting conditions, pull-in effects could be avoided also when the control flow adaptation operators are invoked during reactive adaptation. However, there are two reasons why the control flow operators try to minimize execution delays and thus push-out effects, but not pull-in effects:

- First, push-out effects are viewed as more worse than pull-in effects. This is because a push-out effect always means an execution delay of nodes, and thus that a workflow may violate time tables and deadlines. This negative side-effect is independent from the question whether a node pushed out is affected by a control action or not. In contrast to this, pull-in effects generally have a positive effect, namely that a workflow may be executed faster. We recall from 3.4.4 (*Adaptation Side-Effects*), that a pull-in effect is a problem only if a node that originally would have been executed *beyond* the valid time interval VT of a control action ca is affected by a control action valid during VT after having been pulled into P_{VT} . This will only be the case for a minority of nodes. Thus, pull-in effects should not generally be avoided by inserting waiting conditions to a workflow as shown in Figure 8-23.

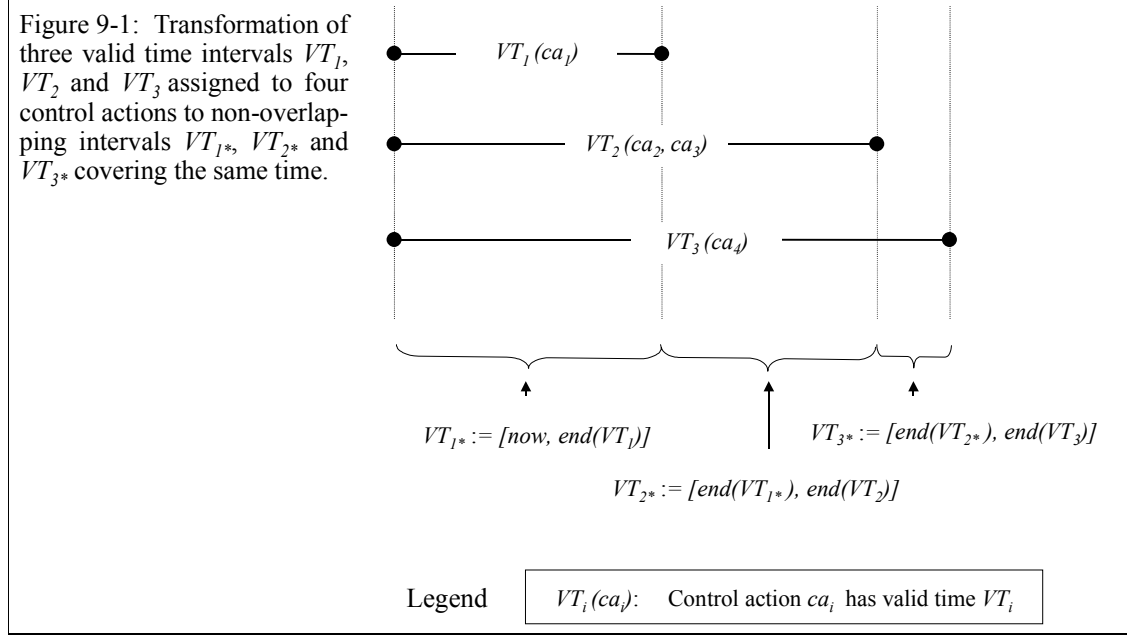
- Second, if the control flow operators would insert waiting conditions to avoid pull-in effects and to obtain free temporal slots to avoid push-out effects w.r.t. further adaptations, these free temporal slots nevertheless generally could not be used for further adaptations if *reactive* control flow adaptation is used. This is because typically such a free temporal slot will already have been consumed when reactive control flow adaptation would notice that it could have made use of the free temporal slot. For example, imagine that in Figure 8-23 the activity definition of node 3 would have to be replaced by an activity definition with a much longer duration. Then, the free temporal slot between node 1 and 3 could be used to avoid an execution delay by starting node 3 with its new activity definition earlier, or – more precisely – by reducing the waiting condition between node 1 and 3 by that duration that additionally is needed for the execution of node 3 with its new activity definition. However, due to the definition of reactive adaptation, the need for such a free temporal slot would be noticed not before node 3 is set to state *Control-Activated*, and thus when the free temporal slot is not available anymore as the edge (1,3) has already been executed. The only constellation where a usage of such a free temporal slot may be possible for reactive adaptation would be if there is a free temporal slot available in a *parallel* path, such as the free temporal slot between node 5 and the AND-JOIN node in Figure 8-23. Then, node 3 with its new activity definition could be moved to this lower path. However, this will be a very seldom constellation which does not justify the general generation of waiting conditions.

These two arguments show that in contrast to push-out effects it does not make very much sense to enhance the introduced control flow operators by mechanisms that are able to avoid pull-in effects independently from the adaptation strategy during which the operators are invoked. Rather, pull-in effects can only be handled in a controlled manner by *predictive* control flow adaptation as this adaptation strategy has an explicit notion of the workflow part P_{VT} corresponding to the valid time VT of a triggered control action (in contrast to the control flow operators introduced in this chapter). Not surprisingly, predictive control flow adaptation also has more possibilities to avoid push-out effects than offered by the control flow operators themselves. These aspects will now be described in Chapter 9 where we describe predictive control flow adaptation in detail.

Predictive Control Flow Adaptation

In this chapter, we describe predictive control flow adaptation. As discussed in 1.3, the main advantage of this strategy is that it gives workflow users more time to prepare themselves w.r.t changed control flow definitions than reactive adaptation. Furthermore, due to the limitations of the control flow operators of Chapter 8, reactive control flow adaptation cannot minimize and control pull-in effects, and provides only limited mechanisms to minimize and control push-out effects. In particular, if *multiple* control actions affect one workflow simultaneously and are handled by reactive adaptation, there is no way to think about suitable orders of processing them in order to minimize pull-in and push-out effects. This is because for reactive adaptation the order in which control actions are translated to structural workflow adaptations is determined by the node execution order, and not by any higher-level algorithm. The consequence is that pull-in and push-out effects cannot be controlled. In contrast to this, predictive adaptation with its explicit notion of the workflow part corresponding to a control action's valid time interval has much more possibilities to minimize and control such pull-in and push-out effects. This will be described in this chapter.

The chapter is organized as follows: In Section 9.1, we introduce some interval transformation to reduce the complexity of the problem. These transformations allow us to assume that different control actions triggered simultaneously have the *same* valid time interval. In Section 9.2, we describe the algorithm for predictive control flow adaptation. In particular, we describe in which *order* AGENTWORK invokes the different control flow operators to minimize and control pull-in and push-out effects. Section 9.3 shows how a workflow that has predictively been adapted is monitored after its continuation, and how an adaptation is corrected if the adaptation assumptions do not match the execution reality. The chapter concludes with a summary and discussion in Section 9.4.



9.1 Transformation of Valid Time Intervals

Generally, different control actions affecting one workflow may have different valid time intervals VT_i . We recall from 7.3 (*Valid Time Conventions for Control Actions*) that a valid time interval assigned to a control action first always starts at *now*, i.e., the moment at which the (local) control flow failure occurred, and second always is coherent, i.e., does not consist of several unconnected parts. Thus, we can write

$$VT_i = [now, end(VT_i)] \quad \text{for } i = 1, 2, \dots \quad (i)$$

which we order w.r.t. their duration (with VT_1 being the interval with the smallest duration). AGENTWORK then transforms these VT_i to valid time intervals VT_{i*} which do not overlap but which together cover the same time. This is done by setting

$$VT_{1*} = [now, end(VT_1)] \quad \text{and} \quad (ii)$$

$$VT_{i*} = [end(VT_{i-1*}), end(VT_i)] \quad \text{for } i = 2, 3, \dots \quad (iii)$$

(Figure 9-1). The control actions are assigned to these new valid time intervals VT_{i*} in a way that the union of a control action's valid time intervals is exactly the original valid time interval in (i). For example, the control actions of Figure 9-1 are assigned to the new valid time intervals as follows:

Interval Control action	VT_{1*}	VT_{2*}	VT_{3*}
ca_1	YES	NO	NO
ca_2	YES	YES	NO
ca_3	YES	YES	NO
ca_4	YES	YES	YES

To fulfill that a valid time interval always has to start at *now*, AGENTWORK first only considers the control actions with valid time interval VT_{1*} (second column in table above), as the two other intervals VT_{2*} and VT_{3*} do not start at *now*. The control actions with valid time interval VT_{i*} ($i = 2, 3, \dots$) interval are considered after $VT_{(i-1)*}$ has elapsed, i.e., when *now* has the value $end(VT_{(i-1)*})$.

By these transformations, we can assume in the following that different control actions triggered simultaneously have the *same* valid time interval VT .

9.2 Algorithm for Predictive Control Flow Adaptation

We now describe the principal algorithm that is performed to translate multiple control actions

$$ca_i \quad VALID-TIME \quad VT$$

with the same valid time interval VT into a structural control flow adaptation. For this, we first describe the goals and principles of the algorithm (9.2.1). Second, we explain details and give an illustrating example (9.2.2).

9.2.1 Goals and Principles

The overall goal of this algorithm is to reduce push-out effects to a minimum (in order to minimize execution delays), to identify and control unavoidable push-out effects, and to control pull-in effects.

The main steps of the algorithm are described now. Note that this algorithm is the same for the different versions of predictive adaptation, i.e., for 1. “one-shot” predictive adaptation, 2. iterative predictive adaptation with sub-intervals, and 3. conditional iterative predictive adaptation (3.4.1). The only difference is that for iterative predictive adaptation with sub-intervals the considered valid time typically is significantly shorter than for “one-shot” predictive adaptation, and that for conditional iterative predictive adaptation untouched conditional paths are excluded from P_{VT} . All execution durations mentioned in the following are assumed to be average execution durations, according to the default average case estimation strategy of AGENTWORK.

Main Step 1: Processing of Non-Pushing Control Action Applications (without Allowing Pull-In Effects)

In this main step, those control actions are processed for those nodes for which the corresponding

control flow operator can be performed in a non-pushing way. By this it is avoided that nodes so far belonging to P_{VT} are not anymore a member of P_{VT} (i.e., that they are “pushed out“ from P_{VT}). In particular, by this the side-effect is avoided that a node n is affected by a control action ca valid during VT but is pushed out from P_{VT} by another control action before ca could have been applied to n .

As the non-pushing application of a control flow operator may cause pull-in effects, the following mechanism is used to suppress these pull-in effects during this main step: Whenever the execution duration of a path within P_{VT} is decreased due to some non-pushing application of a control flow operator, AGENTWORK assigns a temporary *virtual duration* to the path. This virtual duration consists of the remaining in-fact execution duration plus the *free temporal slot* obtained by the non-pushing operator application. For example, if a node with the execution duration of 6 hours is dropped from a path with execution duration of 24 hours (including the execution duration of the node to be dropped), then AGENTWORK would assign to this path the virtual duration of 24 hours, consisting of

$$\begin{aligned} &(18, \text{hour}) \quad // \text{remaining execution duration of path after node dropping} \\ &+ \\ &(6, \text{hour}) \quad // \text{free temporal slot obtained by node dropping.} \end{aligned}$$

On one side, this virtual duration avoids that at this stage nodes originally located beyond P_{VT} are executed during P_{VT} , i.e., are pulled into P_{VT} . On the other side, this virtual duration assigned to an affected path “reserves“ free temporal slots, and thus increases the possibilities for further control flow operators to be applied in a non-pushing way.

Main Step 2: Controlled Processing of Pushing Control Action Applications

The next main step consists of processing those control actions for those nodes for which push-out effects occur. These push-out effects are controlled as follows: For any node that would be pushed out but has been affected by other control actions before the push-out, the user is requested whether such a control action shall still be applied to this node or not.

Main Step 3: Controlled Processing of Pull-In Effects

This main step deals with those paths for which free temporal slots are still available after main step 2, i.e., for which the virtual duration is different from the estimated in-fact duration. These paths have to be considered as after the processing of the control actions no path should consist of any free temporal slot anymore. This is because the reason for these free temporal slots only has been first to avoid pull-in effect during main step 1 and second to insertion intervals for time-consuming control actions such as $add(A, C)$ in order to minimize push-out effects during main step 2. Thus, after main step 2 these free temporal slots have to be consumed by

- a) either pulling nodes into P_{VT} or
- b) by translating the free temporal slots to waiting conditions assigned to control flow edges if pull-in effects shall be avoided.

The decision criteria are the following: For every node n that would be pulled in if the free temporal slots would be deleted, it is checked whether n is affected by any control action ca valid *during* VT . If this is not the case, n is pulled into P_{VT} without user interaction as there is no reason why n should not be executed earlier. If this is the case, the user is requested whether n shall be pulled in. If the user agrees, n is pulled into P_{VT} and ca is applied to n . Alternatively, the user can also decide that n is pulled into P_{VT} , but that ca is not applied to n . If the user disagrees to pull in n , waiting conditions are generated to avoid that n is pulled in, i.e., to achieve that n will be executed beyond VT .

9.2.2 Details and Illustrating Example

We now describe the detailed structure of the algorithm, in particular in which order the particular non-pushing and pushing control action applications are processed. For this, we use the following example (assuming the same valid time interval for all control actions according to 9.1):

- First, for valid time $VT = [now, now + (24, hour)]$ the control actions

$drop(S, C)$	with	$dur-av(S) = (4, hour)$
$postpone(W, (6, hour), C)$	with	$dur-av(W) = (5, hour)$
$postpone(T, (16, hour), C)$	with	$dur-av(T) = (6, hour)$
$replace(X, Z, C)$	with	$dur-av(X) = (8, hour), dur-av(Z) = (4, hour)$
$add(B, C)$	with	$dur-av(B) = (5, hour)$

have been triggered simultaneously.

- Second, AGENTWORK has identified the workflow shown in Figure 9-2 a) as being affected by these control actions (according to 7.4.3). In particular, it has been estimated that the 9 nodes shown in the dashed rectangle (P_{VT} in Figure 9-2 a) will be executed during VT . Furthermore, we assume that the nodes 3 and 9 (i.e., the nodes of the failure node set) are in state *Untouched*.
- Third, we assume that $R \rightarrow T \rightarrow U$ is a logical sequence according to 8.1.3.1.

We emphasize that this an extreme example to illustrate that the algorithm for predictive adaptation can cope with multiple control actions that affect the same workflow part simultaneously. According to assumption 1 (*Limited Number of Simultaneously Triggered Control Actions*) described in 8.1.2, we expect that in real-world applications it will not occur very often that a workflow part consisting only of 9 nodes will be affected by so many control actions simultaneously. Thus, this example should be viewed as a sort of a “worst case” example.

As a preliminary remark, we state that this algorithm uses the control flow operators described in Chapter 8 (*Structural Adaptation Operators*) not as monolithic blocks, but often uses only sub-functionality of them which we assume to be available as sub-operators etc. Furthermore, we omit the *change-value* control action as it cannot cause push-out or pull in effects and thus does not add any relevant complexity to the problem.

Main Step 1: Processing of Non-Pushing Control Action Applications (without Allowing Pull-In Effects)

Step 1.1: Processing of Dropping Control Actions

First, the dropping control actions *drop* and *drop-activities-of* are processed. The simple reason for processing these control actions first is that they reduce the complexity for the following steps as they remove nodes from the workflow. In particular, the application of the corresponding control flow operator *cfop-drop-node* never can cause push-out effects.

For the example of Figure 9-2 this means, that first the control action *drop*(*S*, *C*) is processed, i.e., *S*-node 3 is dropped from the control flow by *cfop-drop-node* (Figure 9-2 b). Then, the virtual duration of the affected path 1 is set to 24 hours, namely

$$\begin{aligned} &(20, \text{hour}) \quad // \text{ execution duration of path 1 after node dropping} \\ &+ \\ &(4, \text{hour}) \quad // \text{ free temporal slot obtained by node dropping} \end{aligned}$$

After having processed dropping control actions, those non-additive control actions are processed during the next steps that affect already existing nodes in a non-pushing manner, but do not add additional activity nodes to a workflow, i.e., postponing and replacing control actions. The algorithm starts with postponing control actions (step 1.2) and then processes replacing control actions (step 1.3). The opposite order (i.e., replacing control actions and then postponing control actions) would be possible as well.

Step 1.2: Processing of Non-Pushing Postponing Control Action Applications

At this step, those *postpone* and *postpone-activities-of* control actions are processed for those nodes for which this can be done in a non-pushing way. This is possible under the following conditions:

1. The postponement duration is not longer than the sum of the free temporal slots of the path to which the affected node belongs to. Thus, a non-pushing *sequential* postponement can be performed by *cfop-postpone-node*.
2. A non-pushing parallel postpone can be performed by *cfop-postpone-node*.

For the *W*-node 10 affected by *postpone*(*W*, (6, hour), *C*) in Figure 9-2 b), condition 1 is not given as there is no free temporal slot for path 2. Thus, it is checked whether a non-pushing parallel postpone can be performed for node 10 which is temporally optimal according to Definition 8.1. This is possible for node 10, as the sum of its duration of 5 hours plus the postponement duration of 6 hours is not longer than the duration of the other parallel path starting at node 11 and ending at node 14 (i.e., not longer than the duration of 17 hours). Thus, node 10 is sourced out to the new parallel path between a new AND-SPLIT node 15 and a new AND-JOIN node 16 (path 3 in

Algorithm for Predictive Control Flow Adaptation

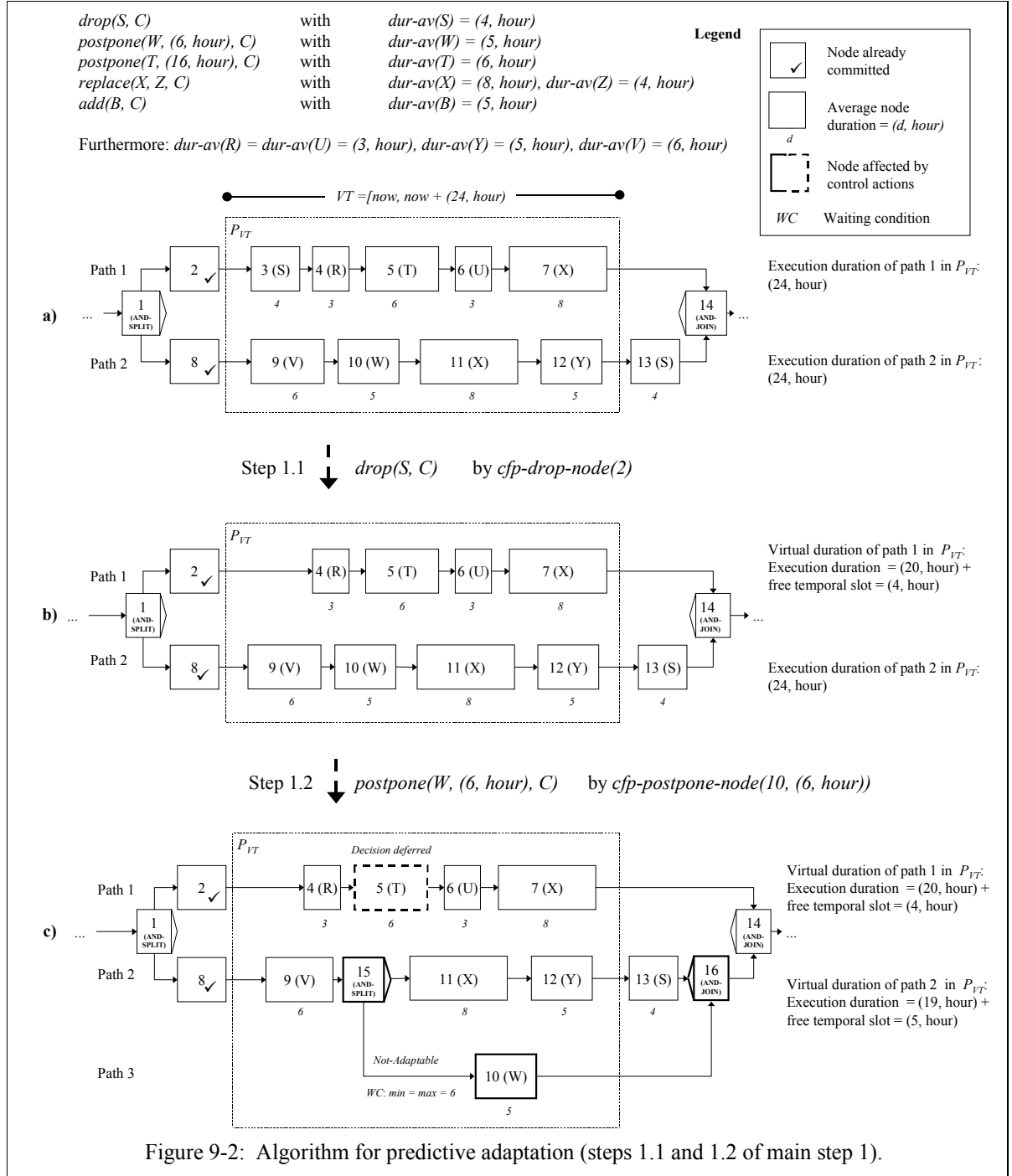


Figure 9-2: Algorithm for predictive adaptation (steps 1.1 and 1.2 of main step 1).

Figure 9-2 c). To realize the postponement of node 10, a waiting condition of duration (6, hour) is assigned to the new edge (15,10). Furthermore, edge (15,10) is marked as *Non-Adaptable* (see 8.1.3.3). This is to avoid that the postponement duration is increased by inserting additional nodes between node 15 and node 10 during subsequent adaptations.

Finally, the virtual duration of the affected path 2 is set to 24 hours, namely

$$\begin{aligned} &(19, \text{hour}) \quad // \text{ execution duration of path 2 after parallel postpone} \\ &+ \\ &(5, \text{hour}) \quad // \text{ free temporal slot obtained by parallel postpone} \end{aligned}$$

For the other postponing control action $\text{postpone}(T, (16, \text{hour}), C)$, a non-pushing processing of the affected T -node 5 is *not* possible. This is because first a non-pushing *sequential* postponement is not possible as for path 1 the duration of the free slot of 4 hours is less than the postponement duration of 16 hours. Thus, if a sequential postponement would be performed, the execution of the successor nodes of node 5 would be delayed by 16 minus 4 hours, i.e., by 12 hours (if the free temporal slot is consumed). In particular, X -node 7 would be pushed out from P_{VT} what shall be avoided at this step because this X -node is affected by the control action $\text{replace}(X, Z, C)$.

Second, parallel postponement cannot be performed as $R \rightarrow T \rightarrow U$ is a logical sequence. Thus, T -node 5 cannot be sourced out to a new parallel path as it has been done for the W -node 10 (assuming that the requested user wants to maintain the logical sequence $R \rightarrow T \rightarrow U$ for this workflow). Therefore, the processing of $\text{postpone}(T, (16, \text{hour}), C)$ is not done during this step, but deferred until main step 2 (*Controlled Processing of Pushing Control Action Applications*).

Step 1.3: Processing of Non-Pushing Replacing Control Action Applications

At this step, those *replace* control actions are processed for those nodes for which this can be done in a non-pushing way. This is possible under three conditions (let A denote the old, B the new activity definition):

1. $\text{dur-av}(B)$ is *not* longer than $\text{dur-av}(A)$, so that a non-pushing sequential replace can be performed by *cfop-replace-act-def*.
2. $\text{dur-av}(B)$ is longer than $\text{dur-av}(A)$, but the sum of free temporal slots is larger than $\text{dur-av}(B) - \text{dur-av}(A)$. Thus, a non-pushing sequential replace can be performed as well.
3. A non-pushing parallel replace can be performed by *cfop-replace-act-def*.

In Figure 9-3 d) the control action $\text{replace}(X, Z, C)$ can be processed in a non-pushing way for X -node 11, as for this node condition 1 holds. This is because $\text{dur-av}(Z) = (4, \text{hour})$ is shorter than $\text{dur-av}(X)$. Thus, *cfop-replace-act-def* is instructed to perform a sequential replace, i.e., for node 11 the activity definition is switched from X to Z . In particular, this increases the free temporal slot of path 2 from 5 hours to 9 hours (while the virtual duration of path 2 in P_{VT} is still 24 hours).

Algorithm for Predictive Control Flow Adaptation

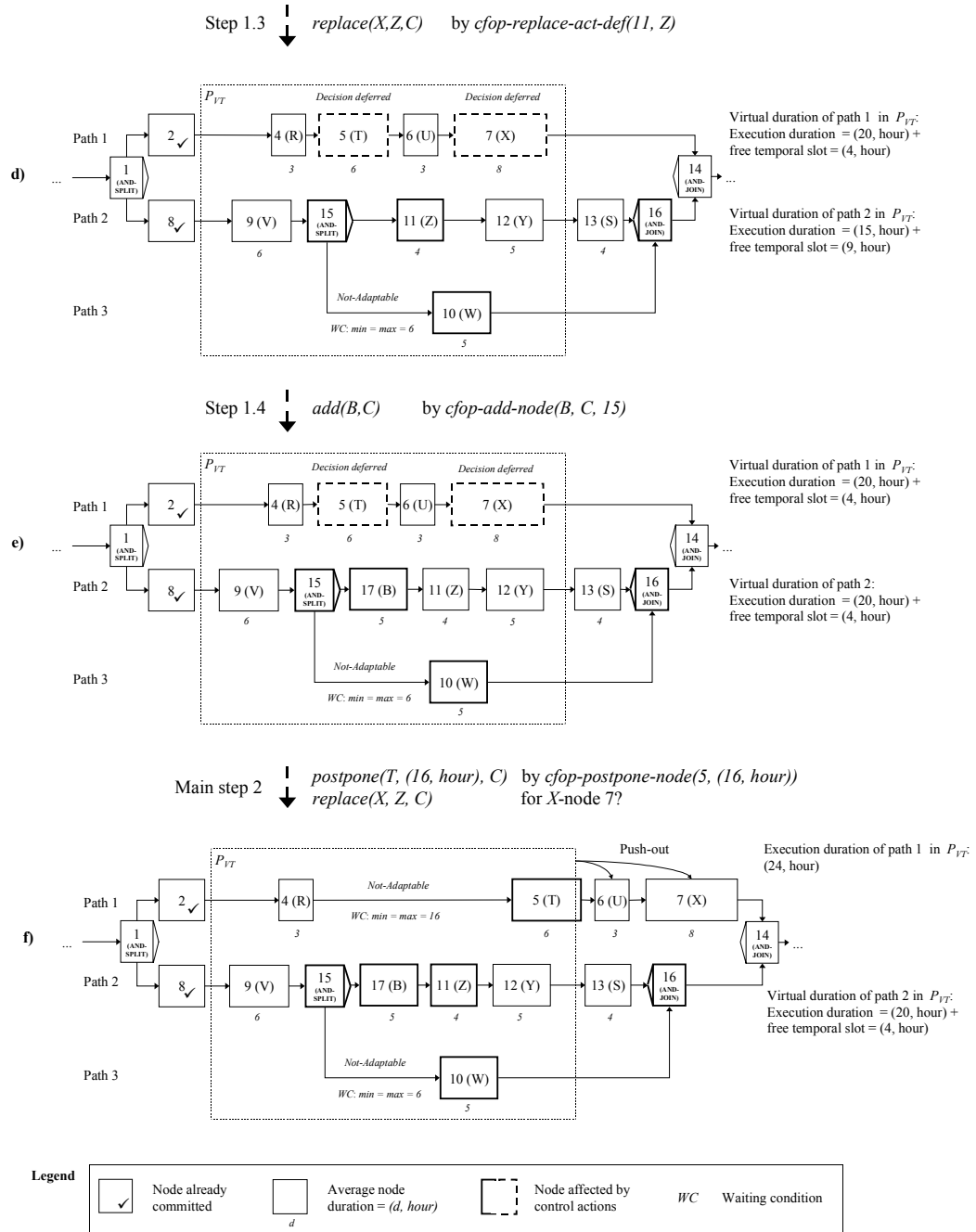


Figure 9-3: Algorithm for predictive adaptation (steps 1.3 and 1.4, main step 2).

Concerning X -node 7, the decision is deferred as it is not yet clear whether this node will be pushed out by the deferred postponing control action $postpone(T, (16, hour), C)$ affecting T -node 5.

Step 1.4: Processing of Non-Pushing Adding Control Action Applications

Fourth, those *add* and *add-repetitively* control actions are processed for which this can be in a non-pushing way. For *add-repetitively*, this always holds as the activity node that shall be executed repetitively is inserted into a new parallel path between the START and END node of the workflow (see 8.2.5). Thus, it cannot push out any node from P_{VT} .¹

For *add*, a non-pushing processing is possible under the following conditions (let A denote the activity definition of the node to be inserted):

1. For at least one path in P_{VT} , the sum of free temporal slots is larger than $dur-av(A)$, and within this path there is at least one insertion point that does not violate an existing logical sequence (or a logical sequence is violated but the user allows to insert the new node nevertheless). Thus, a non-pushing sequential add can be performed by *cfop-add-node*.
2. A non-pushing parallel add can be performed by *cfop-add-node*.

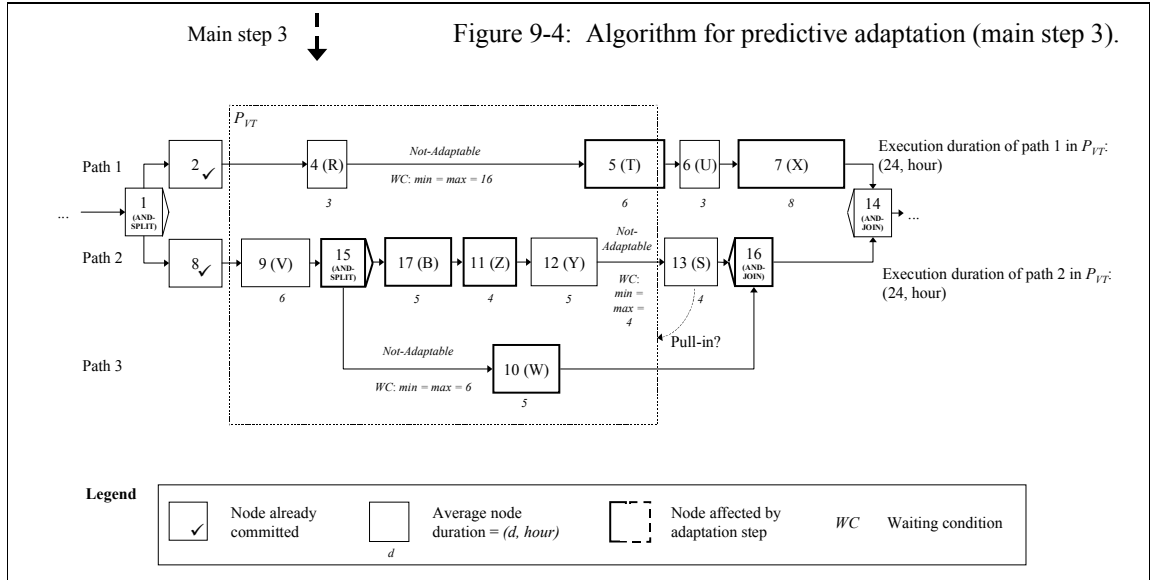
For example, in Figure 9-3 e), $add(B, C)$ can be processed in a non-pushing way, as $dur-av(B) = (5, hour)$, and as the free temporal slot of path 2 is 9 hours. Thus, a new B -node 17 can be inserted into this path. Assuming that path 2 is not affected by any logical sequence, the new B -node can be inserted anywhere between node 8 and node 13. Starting from the insertion position between node 8 and 9, AGENTWORK subsequently offers all insertion points in path 2 within P_{VT} to the user. In our example, the user has selected the position between node 15 and node 11. Thus, $cfop-add-node(B, C, 15)$ is instructed to sequentially insert the B -node at this position.

Main Step 2: Controlled Processing of Pushing Control Action Applications

After steps 1.1 to 1.4, those control actions have been processed for those nodes for which no push-out effects occurred. As sketched in 9.2.1, the next main step (main step 2) then is to process those control actions for those nodes for which push-out effects occur, and to control these push-out effects by requesting the user for any affected node.

For the example in Figure 9-3, this means to process $postpone(T, (16, hour), C)$ for T -node 5 by a sequential postpone, as a parallel postpone is not possible (see step 1.2). Thus, first T -node 5 is postponed by 16 hours by assigning a waiting condition of duration $(16, hour)$ to the edge (4,5) (Figure 9-3 f). This pushes node U -node 6 and X -node 7 out from P_{VT} . The push-out of U -node 6 is uncritical as node 6 is not affected by any control action valid during VT . However, the push-out is critical for node 7 as this node has been affected by $replace(X, Z, C)$ when being a member of P_{VT} . Thus, the user is requested whether the $replace(X, Z, C)$ control action shall be applied to node 7 or

1. Therefore, *add-repetitively* also could be processed at any step of the algorithm, e.g., at the end of it. However, due to its additive semantics it is processed together with *add*.



not. In this example, the user has decided that this shall not be the case, so that node 7 keeps activity definition X when pushed out from P_{VT} .

As a consequence of the push-outs performed during this main step, the temporal free slots are consumed for those paths for which nodes have been pushed out (otherwise there would not have been any push-out). For example, due to the push-out of nodes 6 and 7 in path 1, this path does not have any free temporal slots anymore in P_{VT} . Consequently, the virtual duration of such a path then is the same as the execution duration.

Main Step 3: Controlled Processing of Pull-In Effects

After the push-outs have been processed in a controlled manner by main step 2, main step 3 deals with the remaining paths for which free temporal slots are still available.

In our example (Figure 9-4), path 2 has to be considered, as this path consists of a free temporal slot of 4 hours. Thus, the user is requested whether S -node 13 shall be pulled into P_{VT} with the consequence that it would have to be dropped due to the $drop(S, C)$ control action valid during VT . For this example, we assume that the user does not want node 7 to be pulled into P_{VT} . Thus, to avoid this a waiting condition of duration (4, hour) has to be inserted between node 12 and node 13. Additionally, edge (12,13) is marked as *Non-Adaptable*. As a consequence, the virtual duration of this path then is the same as the execution duration.

9.3 Workflow Monitoring after Predictive Control Flow Adaptation

Predictive control flow adaptation inherently requires that the affected workflow is monitored after the adaptation to check whether the temporal estimations on which the predictive adaptation is

based match the actual execution of the adapted workflow². In 3.4.5 we have identified two principal situations that are relevant for workflow monitoring, namely that a workflow is executed faster than estimated (temporal acceleration), or that it is executed slower than estimated (temporal delay). Now, we want to cover *all* constellations that are relevant for workflow monitoring, in particular those where temporal acceleration and temporal delay overlap, as for example in an AND-SPLIT/AND-JOIN block one executed path is faster and another one slower than estimated. Thus, we formulate the workflow monitoring problem now on a more abstract level, and discuss the specific consequences for the different types of control actions. As we will see, the workflow monitoring problem closely correlates to the problem of pull-in and push-out effects (3.4.4), so that the mechanisms described in 9.2 to handle such effects can also be used for workflow monitoring.

We first formalize the task of workflow monitoring and characterize the meta information required for this task (9.3.1). Then, we describe workflow monitoring for *non-additive* control actions (9.3.2) and for *additive* control actions (9.3.3). In the following, *ca* will denote a control action valid during an interval *VT*. To illustrate the AGENTWORK monitoring approach, we use the resulting workflow of the sample adaptation of 9.2, i.e., the workflow shown in Figure 9-4. As a preliminary remark, we state that all steps described below are subject to user control.

9.3.1 Monitoring Task and Required Meta Information

9.3.1.1 Monitoring Task

Let $P_{VT, est}$ denote the workflow part that has been estimated to be executed during a valid time *VT* after the adaptation (i.e., $P_{VT, est}$ corresponds to the so far used P_{VT} after the completion of an adaptation), and let $P_{VT, in-fact}$ denote that workflow part that is in-fact executed during *VT*. Obviously, the (optimistic) assumption of predictive adaptation is that it holds

$$P_{VT, est} = P_{VT, in-fact} \quad (iv)$$

Then, we can define

Definition 9.1: Monitoring Task

The task of workflow monitoring is to identify as soon as possible whether the following two conditions hold during workflow execution:

- a) $P_{VT, est} \neq P_{VT, in-fact}$ in the sense that there is at least one activity node *n* for which the number of executions is different in $P_{VT, est}$ and $P_{VT, in-fact}$.
- b) The executions of *n* are affected at least by one control action *ca* valid during *VT*.

-
2. For reactive adaptation, monitoring is only required for the advanced check mode (7.4.3.1) as then a node is handled already after it has been set to state *Control-Activated* so that it may happen that it is set to state *Active* beyond the valid time interval of the control action. The monitoring that is required for this is covered by the monitoring for predictive adaptation, so that we do not discuss it further.
-

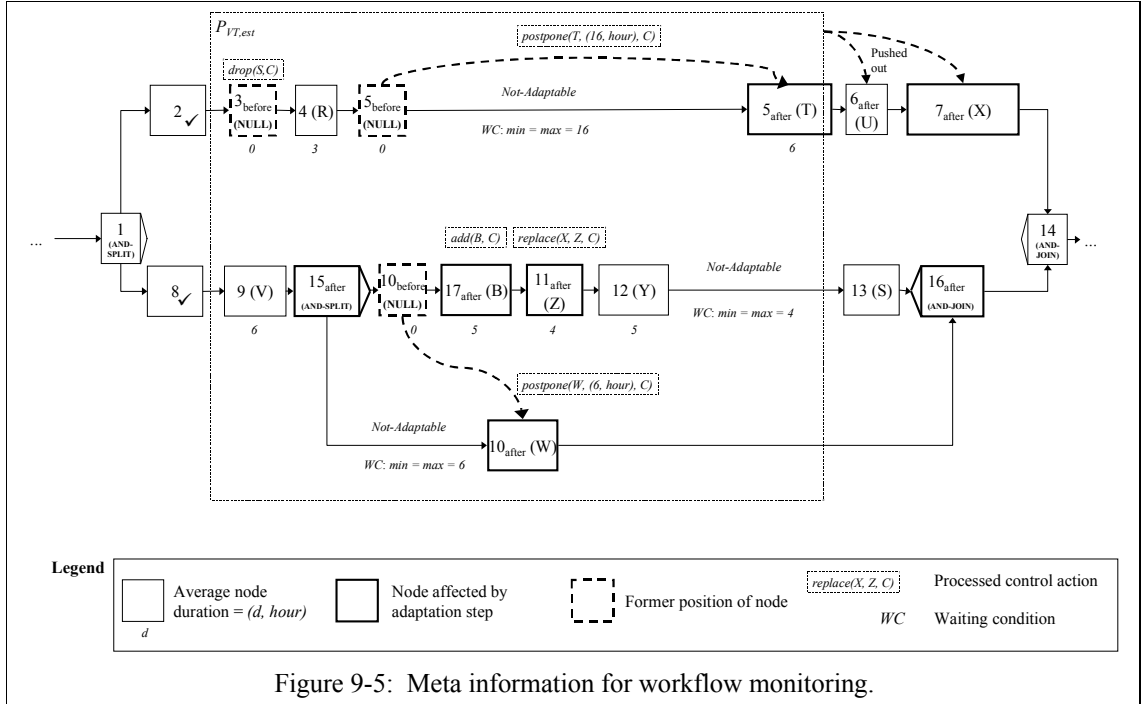


Figure 9-5: Meta information for workflow monitoring.

Possible reasons for $P_{VT,est} \neq P_{VT,in-fact}$ are:

1. A node or edge from $P_{VT,est}$ is executed faster or slower than estimated.
2. A predictive condition resolution (6.4.3) turns out to be wrong.
3. For a loop the assumed number of loop iterations (6.4.4) turns out to be wrong.
4. Due to subsequent control actions triggered after the adaptation, further nodes are added, dropped, postponed, or change their activity definitions.

9.3.1.2 Meta Information for Workflow Monitoring

An important prerequisite for workflow monitoring is that the structure of a workflow before its adaptation is still known, as adaptations may have to be taken back. AGENTWORK manages this by maintaining the following meta information for a workflow that has been adapted: First, NULL-nodes are inserted at the former positions of dropped or postponed nodes (e.g., by inserting NULL-nodes 3_{before} and 10_{before} for the dropped S -node 3 and the postponed W -node 10 in Figure 9-5). Analogously, such a NULL-node is also inserted in case of a parallel replace to indicate the former position of the node of which the activity definition has been replaced. Second, any node affected by a control action (e.g., any node postponed) is labelled (graphically by thick borders as shown in Figure 9-5), and the information of the respective control action is kept.

As indicated in the example of Figure 9-5, n_{before} and n_{after} denote the node identifiers at the position *before* and *after* an adaptation. For dropped nodes, only n_{before} has to be used (as there is *no* new position for such a node). For nodes that are handled by sequential replace or for which only some attribute values have been changed due to a *change-value* control action, n_{before} and n_{after} always are at the same position so that we may use only n as identifier (i.e., $n = n_{before} = n_{after}$). Note that for sequential postponement we have to distinguish between n_{before} and n_{after} to express the temporal “movement” of the postponed node (e.g., see node positions 5_{before} and 5_{after} in Figure 9-5).

9.3.2 Monitoring for Non-Additive Control Actions

For non-additive control actions, we have to distinguish two principal situations:

Situation 1: $n_{before} \notin P_{VT, est}$ but $n_{before} \in P_{VT, in-fact}$ and n affected by ca^3 .

This means that during workflow monitoring it is detected that the estimation that a node n will *not* be executed during *VT* (i.e., $n_{before} \notin P_{VT, est}$) and thus will *not* be affected by ca has been *wrong* (as $n_{before} \in P_{VT, in-fact}$). In other words, it turns out that a node n that has *not* predictively been dropped, postponed etc. is in fact affected by ca as n will be executed *during VT* (i.e., n is “dynamically” pulled into $P_{VT, in-fact}$). Thus, an additional structural adaptation for n may have to be done to satisfy ca . For example, situation 1 may occur if node 13 is pulled into $P_{VT, in-fact}$ due to a faster execution of the sequence $17_{after} \rightarrow 11_{after} \rightarrow 12$. As a consequence, node 13 is affected by $drop(S, C)$ and therefore may have to be dropped as well.

Situation 2: $n_{before} \in P_{VT, est}$ and affected by ca , but $n_{before} \notin P_{VT, in-fact}$

This means that during workflow monitoring it is detected that the estimation that a node n will be executed during *VT* ($n_{before} \in P_{VT, est}$) and thus will be affected by ca has been *wrong* (as $n_{before} \notin P_{VT, in-fact}$). In other words, it turns out that a node n that has predictively been dropped, postponed etc. according to ca is in fact *not* affected by ca as it is executed *beyond VT* (i.e., n is “dynamically” pushed out from $P_{VT, in-fact}$). Thus, the structural adaptation may have to be taken back. For example, situation 2 would occur if node 11_{after} would be set to state *Active* more than 13 hours later than assumed by the estimation on which $P_{VT, est}$ is based (which has estimated that node 11_{after} will be executed after 11 hours (starting from node 9), as this is the average duration needed to execute the predecessor nodes). A reason for this may be that the execution of path $9 \rightarrow 15_{after} \rightarrow 10_{before} \rightarrow 17_{after}$ unexpectedly takes 25 hours instead of 11 hours, so that node 11 is executed 14 hours later than assumed and thus is not anymore a member of $P_{VT, in-fact}$.

To handle these two situations, one could argue that it is appropriate to automatically perform further adaptations operations to satisfy ca in case of situation 1 (e.g., to automatically drop S -node 13 from the workflow), resp. to take back the adaptation in case of situation 2 (e.g., to automatically reassign the old activity definition X to node 11). However, as these situations can be viewed as

3. Strictly seen we do not have to take the particular node, but the particular node *execution*, according to 7.4.3 (affected activity node execution). To avoid a technical overhead, we omit this distinction which is only relevant in case of loops or rollbacks.

some sort of “dynamic” pull-in and push-out effects (i.e., pull-in and push-out effects during workflow execution), the problems discussed for “static” pull-in and push-out effects (i.e., pull-in and push-out effects during workflow adaptation) in 3.4.4 hold analogously. In particular, there it has been discussed that it highly depends on the particular workflow whether such a pull-in or push-out should be allowed or not so that a user should be requested for this. This was also reflected in the main steps 2 (controlled processing of pushing control action applications) and 3 (controlled processing of pull-in effects) of the algorithm for predictive control flow adaptation, where a user is requested to decide whether a push-out resp. a pull-in shall be performed or not.

The dynamic pull-in or push-out effects of situation 1 resp. situation 2 during workflow monitoring are handled analogously, i.e., the user is requested whether the dynamic pull-in (and thus further adaptation operations to satisfy *ca*) resp. the dynamic push-out (and thus the taking back of adaptation operations) shall be allowed and performed. Because of the assumption of a limited number of simultaneously triggered control actions (8.1.2) this user interaction can be viewed as acceptable.

If the user allows the pull-in or push-out (e.g., allows to pull in node 13 in Figure 9-5), the control flow operators described in Chapter 8 are used to perform further adaptation operations being necessary to satisfy *ca* resp. to take back adaptation operations.

9.3.3 Monitoring for Additive Control Actions

For additive control actions, workflow monitoring is easier. First of all, workflow monitoring does not have to consider an *add-repetitively*(*A*,*d*,*C*) control action at all. This is because such a control action is always translated into a loop which is parallel to the rest of the workflow and consists only of an *A*-node which is executed iteratively with period *d* during *VT* (8.2.5). In particular, for translating an *add-repetitively*(*A*,*d*,*C*) control action into structural workflow adaptations *no* estimation is required. Therefore, the possible reasons 1-4 listed in 9.3.1.1 that may lead to $P_{VT, est} \neq P_{VT, in-fact}$ cannot affect the generated loop itself.

Second, for an *add* control action the only relevant situation is that for a new node n_{after} (e.g., the new *B*-node 17_{after} in Figure 9-5) it holds

$$n_{after} \in P_{VT, est} \quad \text{but} \quad n_{after} \notin P_{VT, in-fact} \quad (v)$$

This means that though it has been estimated that the new node n_{after} will be executed during *VT*, it turns out that n_{after} will be executed *beyond VT*. In the example of Figure 9-5, this may occur if the execution of node 9 unexpectedly takes more than 24 hours, so that 17_{after} would not be executed anymore during *VT*.

The specific problem with situation (v) is that it is not sufficient to simply wait until the new node is executed. This is because it then may be detected *too late* that the new node will not be started (i.e., set to state *Active*) during *VT* anymore. For example, if workflow monitoring simply would wait until the nodes 9 and 15_{after} have been executed, and if the execution of node 9 would take more than 24 hours, then it would be detected too late that the control action

$$add(B,C) \text{ VALID TIME } [now, now + (24, hour)]$$

is violated, as B -node 17_{after} is not started anymore during VT ⁴. This is handled as follows:

Let $dur-av_I(n_{after})$ denote the estimated average duration of the data activation phase of the new node n_{after} (according to 6.3.2.1). Then AGENTWORK waits until the interval

$$VT' = [begin(VT), end(VT) - dur-av_I(n_{after})]$$

has expired. If n_{after} then has not yet been set to state *Control-Activated*, this means that the new node runs into danger not to be set into state *Active* during VT from that moment on (as it has to be assumed that the remaining time is needed for the data activation phase). Thus, AGENTWORK first tries to move the new node to a parallel path where the node can be set to state *Control-Activated* immediately.⁵ If such a repositioning is not possible as for example no parallel paths exist, an authorized user has to be informed that the new node may not be started anymore during VT .

9.4 Summary and Discussion

In this chapter, we have described predictive control flow adaptation. In particular, we described in which *order* control actions are processed to minimize and control pull-in and push-out effects much more than this is possible for reactive adaptation. Furthermore, we have described how a workflow that has predictively been adapted is monitored after its continuation, and how an adaptation is corrected if the adaptation assumptions do not match the execution reality.

The central point of criticism that can be made is that predictive adaptation is very complex, and thus may produce uncontrollable adaptation scenarios, in particular when a large number of control actions affects one workflow. In particular, if estimations turn out to be wrong during workflow monitoring and thus adaptations may have to be taken back or have to be performed additionally, the complexity may become overwhelming.

The main counter-argument w.r.t. this point is one already made for the control flow operators of Chapter 8: Due to assumption 1 (*Limited Number of Simultaneously Triggered Control Actions*) introduced in 8.1.2, we do not expect too many control actions (i.e., more than 2 or 3) affecting one workflow simultaneously on the average. Thus, it can be assumed that the complexity remains manageable. Furthermore, as the user is always requested to decide whether nodes affected by control actions may be pulled in or pushed out, one can avoid that adaptations lead to undesirable control flow semantics. Finally, we should emphasize that it is better to support adaptations in a semi-automated manner as provided in this thesis (and to accept the inherent complexity), rather than to disallow such adaptations with the consequence that a workflow system cannot cope with control flow failures at all.

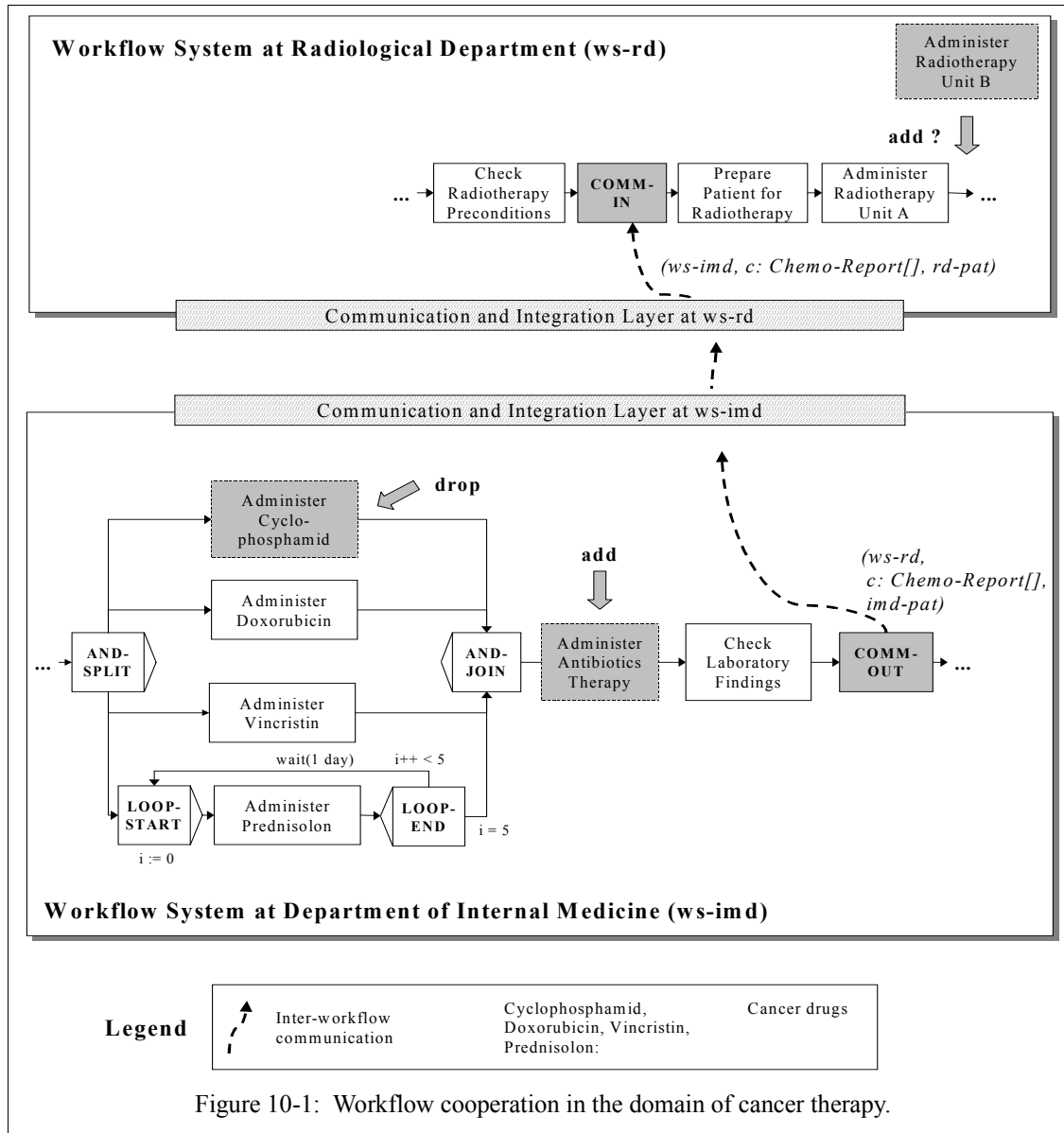
-
4. Recall from 7.2.3.1 that to satisfy the valid time constraint of a control action is it sufficient that the *start* of the node execution (i.e., when it is set to state *Active*) occurs during the valid time.
 5. Note that this parallel path for example may exist due to former adaptations. Note further that at this step AGENTWORK does not try to insert a new parallel path for n_{after} as this would have already been done for the initial insertion of n_{after} if possible (according to 8.2.4).

Handling Control Flow Failures for Cooperating Workflows

As mentioned in Chapter 1, the abortion, suspension or dynamic adaptation of a workflow may affect *other workflows* cooperating with this workflow. Thus, after having described workflow cooperation in Chapter 5, and workflow abortion, suspension and adaptation in Chapter 7 to Chapter 9, we can now address this problem in detail.

The chapter is organized as follows: In Section 10.1, we introduce different types of constraints that can be specified between cooperating workflows and that have to be considered when a control flow failure occurs to one of the cooperation partners. In Section 10.2 and Section 10.3, we describe how it is determined whether a global resp. local control flow failure violates any constraint specified between cooperation partners. In Section 10.4 we describe how a cooperation partner then can cope with the situation that one of its partners cannot meet some specified constraint anymore. The chapter concludes with a summary and discussion in Section 10.5.

Before going into the details, we refer to the example from the domain of cooperative care that has already been sketched in Section 1.3. In this example, a workflow running at the department of internal medicine supports the cancer chemotherapy of a patient (Figure 10-1). The inter-workflow communication definition (see 5.3.10.2) at the COMM-OUT node after the *Check Laboratory Findings* activity specifies that a chemotherapy report (entry *c*: *Chemo-Report[]*) of a patient referenced by *imd-pat* shall be sent to the cooperating workflow system at the radiological department (entry *ws-rd*) after the inspection of laboratory findings. Such a report documents the administered chemotherapy and the current tumor state and is an important base for the treatment at the radiological department. Recall from 5.3.10.2 that *c*: *Chemo-Report[]* is called a communication object as this is the information that has to be communicated.



Vice versa, the cooperating workflow at the radiological department contains a **COMM-IN** node after the *Check Radiotherapy Preconditions* activity with an inter-workflow communication definition. The latter states that at this execution step a chemotherapy report (entry *c: Chemo-Report[]*) for the patient referenced by *rd-pat* is expected from a cooperating workflow system at the department of internal medicine (entry *ws-imd*). A *Chemo-Report* object received from *ws-imd* by the

communication and integration layer at *ws-rd* then has to be forwarded to this COMM-IN node if it holds $imd-pat = rd-pat$ (for this example, we assume hospital-wide patient identifiers which is common practice in most hospitals). The usage of two *separate* workflow systems is motivated by the typically high autonomy requirements of the departments of larger hospitals.

Let us now assume that the workflow at the department of internal medicine is affected by a control flow failure which induces an adaptation, such as that the CYCLOPHOSPHAMID activity is dropped due to some toxicity event, and that an activity supporting the administration of antibiotics is added dynamically to get an infection under control (Figure 10-1). The question now is how this may affect the cooperating workflow at the radiological department. Two principal implication types can be identified:

- *Temporal implications:* Often, a control flow failure will imply that results expected by a cooperation partner cannot be provided anymore in the time frame that originally has been specified between the collaboration partners. In Figure 10-1, the additional antibiotics activity of the workflow at *ws-imd* may cause that the chemotherapy report is delivered later than originally specified. This has the consequence that the radiotherapy activities after the COMM-IN node of the workflow at *ws-rd* may be started later, too. To express such time frames, we will introduce temporal constraints in Section 10.1.1.
- *Qualitative implications:* A cooperation partner usually expects that a result will not only arrive in time but also will meet specific quality criteria. In our medical example, the dynamic dropping of the CYCLOPHOSPHAMID activity may impact the radiotherapy workflow in the sense that an additional radiological unit may become necessary to compensate the cancellation of this drug and to insure tumor remission. Thus, the deletion of activities from one workflow can make it necessary to insert additional activities into a cooperating workflow. In order to express qualitative agreements between cooperation partners, we will introduce qualitative constraints in Section 10.1.2.

It should be emphasized that AGENTWORK does not provide an entirely automated handling for *all* inter-workflow implications of control flow failures. Especially w.r.t. qualitative implications the situation may become arbitrarily complex. Thus, AGENTWORK intends to identify and handle at least the most important types of temporal and qualitative inter-workflow implications.

Note that we address temporal and qualitative implications of control flow failures only in the inter-workflow context, as *intra*-workflow implications either have been already addressed by other authors (e.g., by [DADAM ET AL. 2000] for temporal implications and deadline management), or are assumed to be directly evident for workflow users. For example, w.r.t. our example of Figure 10-1 it can be assumed that the quality-related consequences of dropping an important drug is evident for physicians (but not for the remote radiological colleagues), so that reporting the intra-workflow implications of the drug dropping to the physicians can be viewed as superfluous. However, if this assumption does not hold, the approach described in the following can be easily extended with deriving and reporting the temporal and qualitative implications of control flow failures in an intra-workflow manner.

10.1 Temporal and Qualitative Workflow Constraints

A control flow failure of a workflow has inter-workflow implications if it violates any agreements between the cooperation partners. To express such agreements, AGENTWORK allows to assign *temporal* and *qualitative* constraints manually to workflows. In the following two subsections, we describe how these types of constraints can be assigned to COMM-OUT nodes to specify in which time frames and which quality range results have to be sent to cooperation partners at this COMM-OUT nodes. Analogously, such constraints can also be assigned to COMM-IN nodes so that a cooperation partner knows what it expects from other partners. However, for the problem of inter-workflow implications of control flow failures only constraints assigned to COMM-OUT nodes are relevant so that we concentrate on these.

10.1.1 Temporal Constraints

In AGENTWORK, temporal constraints consist of *deadlines* and acceleration resp. delay *thresholds* that can be assigned to a COMM-OUT node.

A deadline consists of an *absolute* (calendar) point in time of the used time axis (see 4.3.1) which is assigned to a COMM-OUT node at workflow start time or during execution. Relative points in time can also be assigned to a COMM-OUT node at workflow execution time (e.g., node n should be reached 3 weeks after workflow start), but are directly converted to absolute points in time by the system. The possibility to assign relative points in time at workflow *definition* time to express for example that a node should always be reached 3 weeks after workflow start is not supported. This is because AGENTWORK assumes that deadlines are specific for a workflow *instance* and thus should be assigned individually to workflow instances.

Furthermore, to a COMM-OUT node two thresholds *acc-threshold* (*acc* for acceleration) and *delay-threshold* of the duration type (see 4.3.2) can be assigned. If *acc-threshold* and *delay-threshold* are not specified by the user, *acc-threshold* by default is set to ∞ , and *delay-threshold* to zero (i.e., any acceleration and no delay is accepted). The semantics of these thresholds is as follows:

- If an absolute point in time *apt* (such as 20 Jul 2001, 6 pm) has been assigned to the COMM-OUT node as deadline, *acc-threshold* and *delay-threshold* specify that the workflow containing this COMM-OUT node should send its information within the interval

$$[apt - acc\text{-}threshold, apt + delay\text{-}threshold], \quad (i)$$

e.g., within $[20\text{ Jul }2001 - (2, \text{ day}), 20\text{ Jul }2001 + (3, \text{ day})]$. Whenever a control flow failure implies that this will not be possible anymore, the cooperation partner has to be informed.

- If no absolute point in time has been assigned to the COMM-OUT node, *acc-threshold* and *delay-threshold* refer to the relative change in the execution time due to a control flow failure. Let d_{before} denote the execution time that would have been needed to reach the COMM-OUT *before* the control flow failure, and d_{after} the execution time that will be needed to reach the COMM-OUT *after* the control flow failure. The cooperation partner then has to be informed if:

$$\begin{aligned}
d_{before} - d_{after} &> acc\text{-}threshold && \text{(workflow accelerated by more than } acc\text{-}threshold\text{), or (ii)} \\
d_{after} - d_{before} &> delay\text{-}threshold && \text{(workflow delayed by more than } delay\text{-}threshold\text{). (iii)}
\end{aligned}$$

The mechanisms that check whether (i)-(iii) are violated by some control flow failure are described in Section 10.2 and Section 10.3.

We emphasize that the described semantics of these deadlines and thresholds serve the specific purposes of control flow failure handling and their inter-workflow implications. For handling deadlines and temporal thresholds for “normal” workflow execution (i.e., execution not disturbed by control flow failures) we refer to [DADAM ET AL. 2000, EDER ET AL. 1999 A, KAFEZA & KARLA-PALEM 1999, PANAGOS & RABINOVICH 1996]. In particular, these authors describe approaches that allow to assign deadlines and thresholds to arbitrary node types (e.g., activity nodes), and that verify by workflow estimations whether a deadline can be met at all w.r.t. the underlying workflow definition.

10.1.2 Qualitative Constraints

To a communication object o of a COMM-OUT node n , *quality constraints* can be assigned. Formally, such a quality constraint is a F-Logic formula (see 4.2.3) on o that has to be true when workflow execution reaches n . For example, let us assume that the chemotherapy report object c of Figure 10-1 may have different subsections for the applied drugs, for clinical findings and for laboratory findings. Then, by assigning a quality constraint to c such as

$$\begin{aligned}
c.\textit{subsection-for-applied-drugs} &\neq nil && \text{AND} \\
c.\textit{subsection-for-laboratory-findings} &\neq nil,
\end{aligned}$$

both cooperation partners could fix the agreement that in the report at least the subsections for the applied drugs and the laboratory findings may not be empty (i.e., may not be *nil*) as otherwise the radiotherapy workflow cannot continue because important patient data are missing.

Additionally, in many domains the quality of a result can be expressed by a numerical threshold value. For example, the weighted sum of the report’s drug dosages describes the quality of the chemotherapy as it closely correlates to the degree of tumor remission¹. The cooperation partners then could also assign a quality constraint such as

$$c.\textit{weighted-sum-of-drug-dosages} > 100 \text{ mg} \quad (iv)$$

to the transferred report c . If this constraint is violated because some drugs had to be dynamically dropped from the chemotherapy workflow, the radiological department has to be informed as it may be necessary to dynamically add some radiotherapy units to compensate the reduced chemotherapy (as shown in Figure 10-1). Generally, we will call an object that is used to measure the quality of a result as in (iv) a so-called *quality-measuring object*. Another medical example for a quality-measuring object is the weighted sum w.r.t. the degree of negative side-effects², which are

1. The sum is weighted as the different drugs have a different strength w.r.t. tumor remission.

documented in such a chemotherapy report as well. Non-medical examples for quality-measuring objects and constraints on them could be price ranges for e-business interactions or credit limits for banking applications.

Determining how control flow may influence such a quality-measuring object requires additional quality-related knowledge w.r.t. workflow activities. Therefore, in AGENTWORK so-called quality transformation rules can be assigned to an activity definition A stating how A -activities transform a quality-measuring object. For example, to the activity definition

$$A = \text{Drug-Infusion}[drug\text{-}name = \text{VINCRISTIN}, dosage = 2\text{ mg}]$$

quality transformation rule

$$c.\text{weighted-sum-of-drug-dosages} += 2\text{ mg} \quad (v)$$

can be assigned to account for the respective drug dosage increase. At the moment, AGENTWORK supports only constants (e.g., 2 mg) in transformations such as (v), and not for example queries.

Based on this knowledge, qualitative implications of control flow failures can then be determined as we show in Section 10.3.

10.2 Handling Global Control Flow Failures for Cooperating Workflows

We now describe how global control flow failures, i.e., workflow abortions and suspensions, are handled for cooperating workflows.

10.2.1 Handling Workflow Abortions for Cooperating Workflows

In case of a workflow abortion, for every inter-workflow communication definition $(ws; o_1, o_2, \dots, o_n; c)$ of a COMM-OUT node in the remaining control flow a message of the structure

$$(s_1, s_2, \dots, s_n; c; \text{temp-info}) \quad (vi)$$

is sent to ws . The entries s_1, s_2, \dots, s_n describe the states of the objects o_1, o_2, \dots, o_n that have already been reached when the abortion occurred. For example, if o_1 is an expert review needed by the cooperation partner, then s_1 could describe – on an arbitrary granularity level – which topics have already been investigated and which not. The cooperation partner then can decide whether it wants to obtain these already available parts of an object o_i or not. The value of the entry c is the same as in the inter-workflow communication definition and identifies the case. The entry *temp-info* contains the information when the abortion occurred, e.g.,

-
2. For example, the WORLD HEALTH ORGANIZATION provides standardized tables to classify negative side-effects of drug administrations. To each side-effect class a numerical degree is assigned. These tables are used by many hospitals for documentation purposes.

temp-info = abortion of workflow at 27 Jul 2001, 10.00 am.

(vii)

10.2.2 Handling Workflow Suspensions for Cooperating Workflows

In case of a workflow suspension, two situations have to be distinguished w.r.t. the valid time interval VT of the triggering control action, namely that VT is conditional or fixed (see 4.3.4).

10.2.2.1 Conditional Valid Time

If the end of VT is specified by a condition such as *Unless normal-hemato-status(P)*, AGENTWORK cannot inform cooperating workflows for how long the suspension of the workflow will hold (in terms of durations). Thus, nothing more can be done than to send for every inter-workflow communication definition node in the remaining control flow a message of the same structure as in (vi) to the cooperating workflow system. The entry *temp-info* then consists either of the termination condition itself (if this is meaningful for the cooperation partner) or of a heuristic estimation of an authorized user stating for how long the workflow is assumed to be suspended.

10.2.2.2 Fixed Valid Time

If VT is fixed such as $VT = [now, now + (7, day)]$, it is checked for every COMM-OUT node of the remaining control flow whether any temporal constraint is violated by the suspension. This is done

1. by estimating the execution duration of the remaining control flow leading to the COMM-OUT node (i.e., d_{before} as defined in 10.1.1 is estimated),
2. by adding the suspension duration (i.e., the duration of VT) to this duration (to obtain d_{after}),
3. by checking whether any constraint of the structure (i)-(iii) listed in 10.1.1 is violated, and
4. by informing the cooperation partner if such a constraint is violated.

Note that this mechanism does only determine *temporal* implications of workflow suspensions. As a suspension does not change the activity set of a workflow, it cannot have qualitative implications, so that we do not have to consider this type of implication.

However, one problem remains: A COMM-OUT node may be very far away from the failure node set (7.4.1) of the suspending control action, so that the estimation performed in step 1 above may become inherently imprecise. The question how to deal with this problem cannot be answered in general, as the answer depends first on the quality of temporal knowledge about activity and edge execution durations, and second on the average duration and complexity of workflows. Nevertheless, two principal strategies can be identified to deal with this problem:

- First, for every COMM-OUT node the steps 1-4 are performed, and imprecise estimations are detected by workflow monitoring. This strategy has the advantage that it handles all COMM-OUT nodes in a uniform manner. However, it has the disadvantage that due to increasingly imprecise estimations for COMM-OUT nodes far away from the failure node set, messages

about constraint violations sent to cooperation partner often may turn out to be wrong and thus may have to be corrected.

- Second, steps 1-4 are only performed for COMM-OUT for which the temporal distance from the failure node set does not exceed some temporal threshold. For example, for the medical workflow application HEMATOWORK (1.5) it makes sense to consider only COMM-OUT nodes that will be executed not later than two weeks (according to an estimation) after the moment when the suspending control action has been triggered. This is because it has been identified that for this time frame estimations are quite precise for this medical application, while estimations exceeding this time frame often are imprecise. Generally, an appropriate value for such a “maximal estimation” threshold is application-specific and has to be determined heuristically. The advantages and disadvantages of this strategy are diametrical to the other strategy.

The question which of the two strategies is more appropriate depends on the workflow application and can only be answered empirically.

10.3 Handling Local Control Flow Failures for Cooperating Workflows

When a structural workflow adaptation has been performed due to a local control flow failure, it has to be checked whether the affected workflow contains any COMM-OUT nodes in its remaining control flow. If this is the case, it has to be checked whether the applied adaptation affects any cooperation partner. We distinguish between temporal and qualitative implications.

10.3.1 Determining Temporal Implications

The determination of temporal implications is done in the following steps (Figure 10-2):

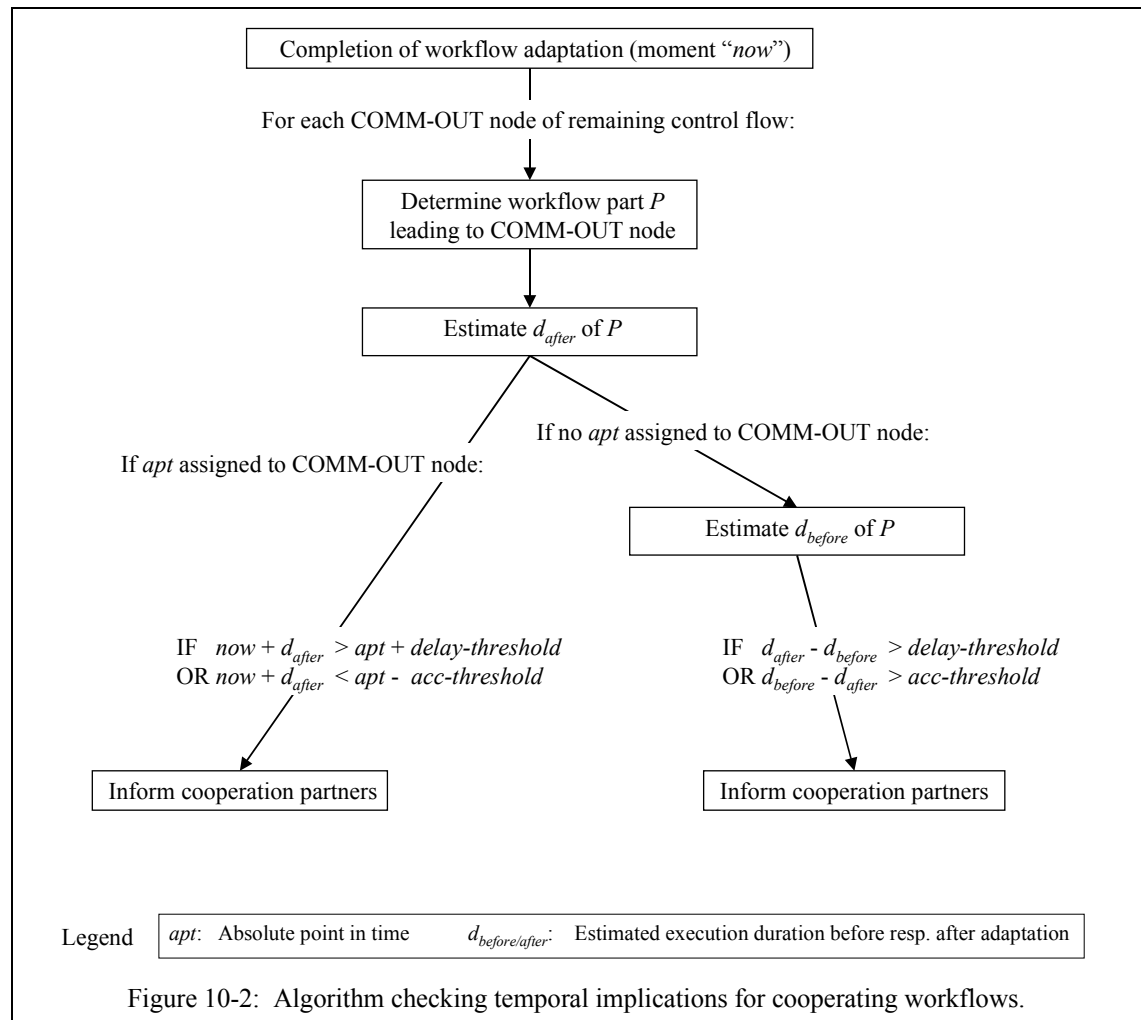
1. For every COMM-OUT node of the remaining control flow, the execution duration d_{after} needed to reach this COMM-OUT node is estimated.
2. If an absolute calendar point in time a_{pt} is assigned to the COMM-OUT node, it is then checked whether this COMM-OUT node can still be reached until this deadline (under consideration of the delay and acceleration thresholds). If the deadline cannot be met anymore due to the adaptation, the cooperation partners are informed (left branch of Figure 10-2).
3. If *no* absolute point in time is assigned to the COMM-OUT node, the duration d_{before} that would have been needed to reach the COMM-OUT node before the adaptation is estimated as well. Then, it is checked whether there is a mismatch between the durations d_{after} and d_{before} violating any temporal threshold of the COMM-OUT node. If this is the case, the cooperation partners are informed as well (right branch of Figure 10-2).

Note that analogously to the situation discussed for workflow suspension, a COMM-OUT node may be very far away from the failure node set of the control action triggering the adaptation, so that the estimation performed in step 1 above may become inherently imprecise. Analogously to

the discussion for workflow suspension, this has to be handled either by workflow monitoring to detect imprecise estimations retrospectively, or by limiting estimations to some maximal temporal interval to minimize the number of imprecise estimations.

10.3.2 Determining Qualitative Implications

For qualitative implications, a modification of the temporal algorithm of Figure 10-2 is used. Instead of estimating the *durations* of workflow parts, AGENTWORK determines the *qualitative* effects of the activities by using the quality transformation rules introduced in Section 10.1.2. Then, AGENTWORK checks whether the derived quality w.r.t. the adapted workflow violates any *quality* constraints (instead of temporal) assigned to communication objects of the COMM-OUT nodes. If



no qualitative implications can be determined because of missing quality transformation rules, the cooperation partners are at least informed which activities have been dropped or added due to the dynamic adaptation. However, as this requires that the activities performed by one workflow are *meaningful* for the cooperation partner, AGENTWORK views this only as an “emergency solution”.

10.4 Handling of Constraint Violations by Affected Cooperation Partners

The question remains how a cooperation partner p can react when it is informed that a specified constraint is violated.

As for p the violation of a specified constraint itself can be viewed as a control flow failure according to Definition 1.1, we assume that p has failure rules stating how to cope with such constraint violations. For example, the workflow system at the radiological department in Figure 10-1 may have a failure rule stating that whenever the chemotherapy dosage applied by the cooperation partner at the department of internal medicine falls below a specified threshold, that then an additional radiotherapy unit has to be inserted into the radiotherapy workflow of the affected patient. Alternatively, an authorized user manually has to decide how to react on a constraint violation.

10.5 Summary and Discussion

In this chapter, we have introduced an approach to deal with control flow failures for cooperating workflows. The approach allows to assign temporal and qualitative constraints to communication nodes so that cooperation partners can fix in which time frame and quality range results should be provided. If a workflow is aborted, suspended or dynamically adapted, it is checked by workflow estimations and the application of quality transformation rules, whether agreed-on temporal and qualitative constraints are violated by the control flow failure. If this is the case, such constraint violations are immediately communicated to affected cooperating workflow systems. The affected cooperation partner then can handle such a constraint violation manually or by failure handling rules stating how to abort, suspend, or adapt its own workflows to cope with the new situation. By this approach, the frequency of failure situations inducing workflow abortion, suspension or workflow adaptations but *not* reported timely to affected cooperation partners can be reduced.

The main limitation of the described approach is that it requires a lot of meta knowledge. Not only temporal knowledge about activity and edge execution durations is needed, but also quality related knowledge such as quality-measuring objects and quality transformation rules. One may argue that such quality-related knowledge often will not be available. The counter-argument is that in many domains quality management guidelines are increasingly used which specify how to measure the quality of products or services. In particular, in many real-world cooperation scenarios there will be at least one object (such as a document) containing information which in some way measures the quality of products or results provided by the cooperation partners. Beside the medical domain, this holds especially for banking domains (credit metrics [SAUNDERS 1999, CAOUETTE 1998]) and insurance business ([ABROMOVITZ & ABROMOVITZ 1997]). Thus, it can be assumed that quality-related knowledge is available for many workflow applications.

During the work on this thesis, large parts of AGENTWORK have been implemented prototypically by the author and several graduate students [NEUBERT 1999, BÖHME 2000, DIETZSCH 2000, GREINER 2000]. In particular, the workflow definition and execution layer and the layer for handling control flow failures (except the inter-workflow agent) have been implemented. Concerning the communication and integration layer, the principal implementation problems have been solved but the implementation still has to be completed in future project phases.

This chapter which describes the AGENTWORK prototype is organized as follows: In Section 11.1, we list the principles on which our implementation is based. In Section 11.2, we describe aspects of the implementation of the workflow definition and execution layer, such as how workflow instances are represented and executed by the workflow engine. In Section 11.3, we describe the CORBA-based implementation approach of the communication and integration layer. In Section 11.4, we describe implementation issues of the layer for handling control flow failures. The chapter concludes with a summary in Section 11.5.

We do not describe the implementation of the application project HEMATOWORK, as this is beyond the scope. For this, we refer to [MÜLLER ET AL. 1998, MÜLLER & HELLER 1998] and several diploma theses which have implemented parts of HEMATOWORK [BRÜMMER 1997, JÖDECKE 1997, FIEBIG 1999]. The entire implementation of HEMATOWORK is subject of a research project currently performed by the University of Leipzig and funded by the German Research Association (DFG). Nevertheless, we sometimes will refer to HEMATOWORK examples to illustrate some implementation aspects of AGENTWORK, such as how the communication and integration layer manages the connection between AGENTWORK and application databases.

11.1 Implementation Principles

The implementation of AGENTWORK is based on the following principles:

1. *Implementation of a Workflow Management System “from Scratch”*

The adaptation approach introduced in the preceding chapters requires a broad range of specific functionality for the workflow management system. For example, the workflow editor has to allow for the specification of estimation values for activity and edge execution durations. At workflow execution time, the internal representation of workflow instances has to support their dynamic adaptation and consistent continuation after the adaptation. As discussed in Chapter 2 (*Related Work*), no commercial workflow management system provides sufficient support for these requirements. In particular, requests to different workflow vendors to obtain such components *and* to achieve an opening of their interfaces failed. Thus, for the workflow definition and execution layer an own, adaptation-oriented workflow editor and workflow engine had to be implemented “from scratch”. To keep the implementation realizable, several components that do not play a central role for handling control flow failures (e.g., worklist handler or organization modeler) have been implemented only in a rudimentary way.

2. *Mapping of ACTIVETFL to Low-Level Programming Languages*

As described in Chapter 4 (*Data and Rule Definition with ActiveTFL*), the AGENTWORK specification language ACTIVETFL is an extension of the object-oriented F-Logic with temporal and active elements. The analysis of available F-Logic implementations (e.g., FLORID [FROHN ET AL. 1997], FLORA-2 [YANG & KIFER 2001], TFL [CARSI ET AL 1998]) showed that none of them can be used for an ACTIVETFL implementation. The main limitation of all of these implementations is that they do not provide sufficient API (Application Programming Interface) capabilities. For example, FLORID only allows to export the final rule processing results into files. Thus, for AGENTWORK components it is not possible to intervene during rule processing (e.g., for user confirmations). Furthermore, control actions derived after rule processing would have to be communicated via file transfers.

Thus, due to these limitations of available F-Logic implementations, ACTIVETFL constructs are mapped to general-purpose programming languages. In particular, the object-oriented core (e.g., class definitions, objects, and object extensions) of ACTIVETFL is mapped to C++, while predicates and rules are mapped to CLIPS [GIARRATANO & RILEY 1993], which is a rule-based programming environment for C/C++.

3. *Middleware-Based Approach for Integration into Distributed and Heterogeneous Environments*

As described in Chapter 3 (*AgentWork Overview*), it is the task of the communication and integration layer to mediate between the high-level logic-oriented ACTIVETFL view of the workflow definition and execution layer and the layer for handling control flow failures on one side, and a distributed and heterogeneous environment on the other side. As already identified by workflow vendors (e.g., IBM MQSERIES WORKFLOW [IBM 2002 C], HEWLETT PACKARD WORKFLOW [SHAN ET AL. 1997]) and research groups (e.g., [MILLER ET AL. 1998, OMG 1998, SCHULZE 1999, WESKE 1999 A]), the usage of a *middleware* such as CORBA or DCOM¹ is of

great usage for the integration of a workflow system into distributed and heterogeneous environments. This is because the location and physical organization of data sources, application programs and other network components can be made transparent for application programming. Thus, it has been decided to use such a middleware, namely CORBA, to implement the communication and integration layer. The specific reasons why CORBA and not some other middleware has been used are given in the respective Section 11.3.

11.2 Workflow Definition and Execution Layer

In this section, we describe implementation aspects of the workflow definition and execution layer, in particular of the workflow editor (11.2.1) and the workflow engine (11.2.2). All components of the workflow definition and execution layer have been implemented in MS VISUAL C++ (version 6.0), for data storage such as the storage of workflow definitions DB2 (version 6.1) has been used.

11.2.1 Workflow Editor

With this component, workflow definitions can be specified in a graphical manner. For example, a workflow's control flow can be specified by drawing control flow edges between activity nodes. The workflow editor consists of two subcomponents, namely the class and activity editor, and the workflow control and data flow editor (Figure 11-1).

11.2.1.1 Class and Activity Editor

This subcomponent is used to specify first the classes of the global ACTIVETFL data schema introduced in 4.2.1, and second to specify activity definitions on the basis of these classes. For example, in the screenshot shown in Figure 11-1 (1), the user has specified the activity definition of a computer tomography (CT) examination, which needs a radiological report as input, focuses on the anatomical area described by this input report, and provides a CT report as output. Furthermore, it has been specified that the application program *CT-Controller* supports this activity (2).² According to Chapter 6, for every activity definition the assumed average, maximal, and minimal execution duration can be specified for workflow estimation purposes, as shown in Figure 11-1 (3).

11.2.1.2 Workflow Control and Data Flow Editor

Based on the class and activity specifications provided by the class and activity editor, the workflow control and data flow editor allows to connect activity definitions by control and data flow edges, as shown in Figure 11-1 (4). In particular, a verification component (5) checks whether the definition constraints introduced in Chapter 5 (*Workflow Definition and Execution*) – such as the block-oriented control flow structure – are met.

1. Distributed Component Object Model

2. Note that *CT-Controller* is a *logical* reference to an application program (5.3.3.2). The connection to the *physical* program (e.g., the “.exe“ file) is maintained by the communication and integration layer.

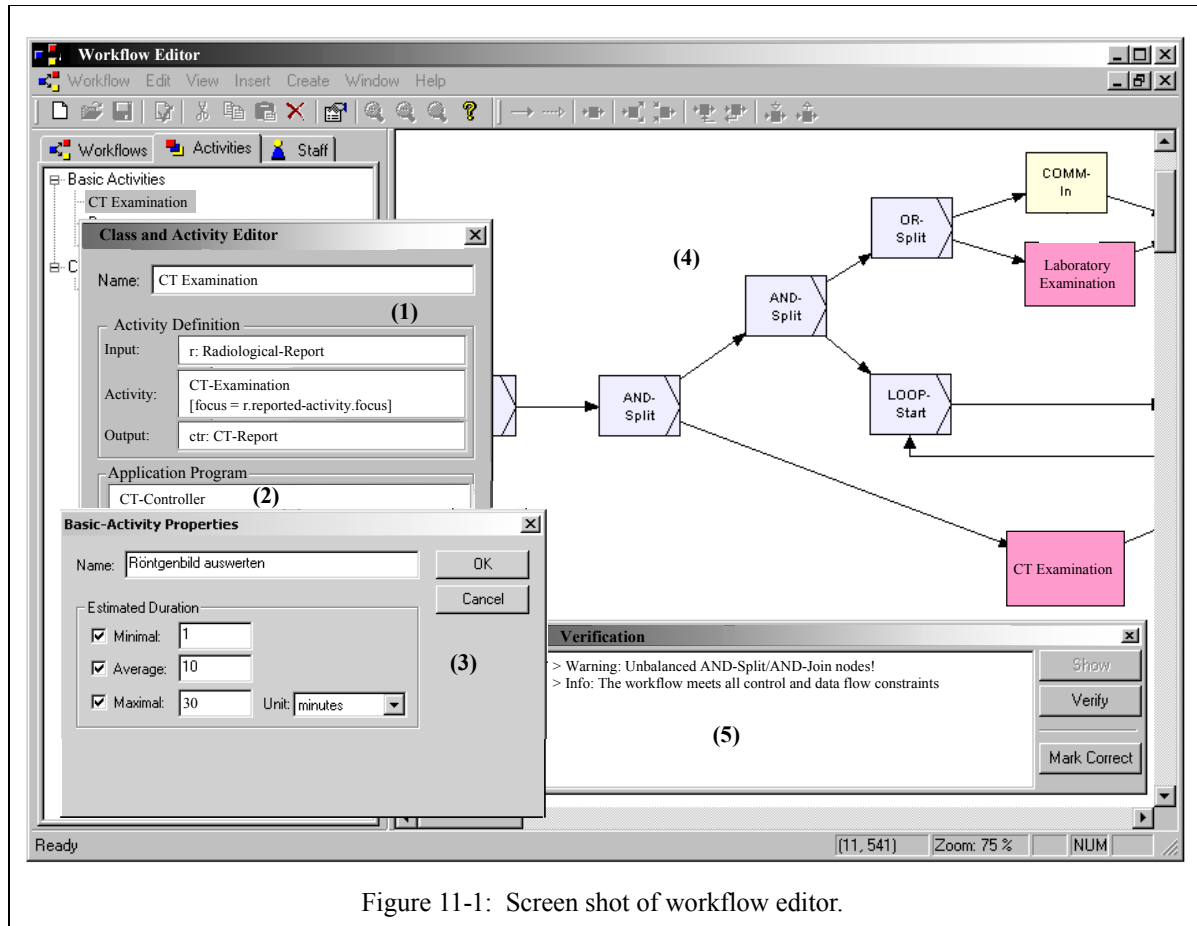


Figure 11-1: Screen shot of workflow editor.

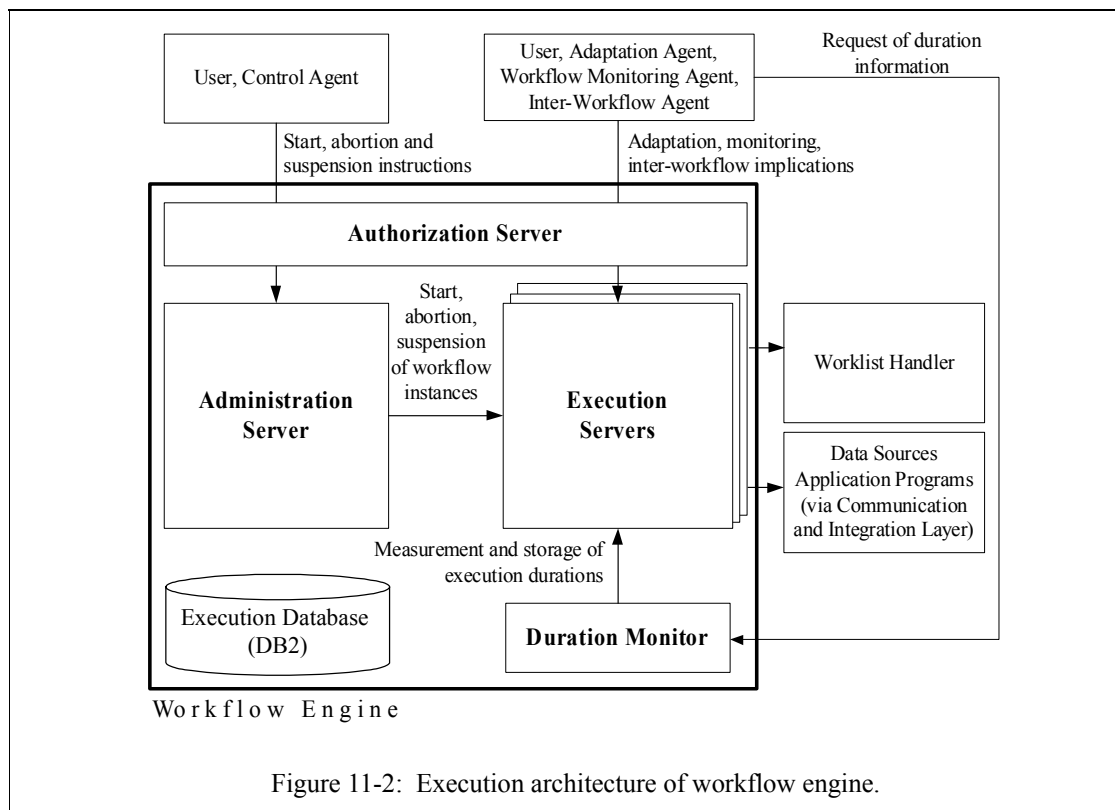
More details about the workflow editor can be found in [BÖHME 2000].

11.2.2 Workflow Engine

We now discuss selected implementation issues of the workflow engine, namely the way workflow instances are internally represented (11.2.2.1) and the architecture to execute such workflow instances (11.2.2.2).

11.2.2.1 Internal Representation of Workflow Instances

To support dynamic workflow adaptation, a workflow instance is internally represented as an *interpreted structure*. This means that a workflow graph such as the one shown in Figure 11-1 (4) is translated into an equivalent internal graph representation and interpreted at workflow execution time. In addition to the elements of the workflow definition, the corresponding internal representa-



tion is enhanced with additional information such as activity execution states or the current values of activity input and output objects.

The alternatives to compile a workflow definition into a set of executable programs [MILLER ET AL. 1998] or to decompose it into a set of interacting rules [BARBARÁ ET AL. 1996] are not suitable. This is because it then would be necessary for an adaptation to reconstruct the control and data flow from interacting programs respective rules.

For a more detailed discussion of these and further instance representation approaches such as using (migrating) data objects [IABG 2002], we refer to [REICHERT 2000].

11.2.2.2 Execution Architecture

Our implementation of the workflow engine is based on [LEYMANN & ROLLER 2000]. The main components of the workflow engine are the *administration server*, the *authorization server*, a number of *execution servers*, the *duration monitor* and an *execution database* (Figure 11-2).

The *administration server* takes as input requests to start, abort or suspend workflows. For example, if a workflow has to be started, the administration server generates a workflow instance on the

basis of the respective workflow definition and assigns a unique identifier to it.

In order to execute a generated workflow instance as an interpreted structure according to 11.2.2.1, a hierarchy of *execution server* instances is created: First, a root execution server instance is created (by the administration server) and started to execute the control and data flow of the workflow instance. For each additional parallel path that has to be executed due to AND-SPLIT or OR-SPLIT nodes, a separate execution server instance is generated by the root execution server. If the paths are joined again, their execution servers are terminated, and the root execution server proceeds with the execution. Depending on the nesting structure of the workflow blocks, this is done recursively, i.e., an execution server executing a parallel path may itself create execution server instances if for example the path contains a nested parallel block.

The execution server instances directly communicate with the worklist handler to update the worklists for the users, and with the communication and integration layer to access data sources and application programs.

Each execution server instance is controlled and maintained within an operating system *thread* [TANENBAUM & WOODHULL 1997]. The alternative to control and maintain each execution server by an own *process* is not appropriate as a large number of active operating system processes would result with a high administration effort. In contrast to this, a thread-based implementation is much more efficient as the execution server instances can use the same address space.

According to Chapter 6, the *duration monitor* measures the execution duration of activities and edges to obtain better duration estimation values than it is possible by specifications at workflow definition time.

All relevant information, such as workflow instance state information or measured activity executions, are stored in a DB2-based *execution database*.

The interaction of the workflow engine components with the agents of the layer for handling control flow failures is as follows: In case of a *global* control flow failure, the control agent instructs the administration server to abort or suspend a workflow instance. In case of a *local* control flow failure, the adaptation agent directly requests the control and data flow definition of the affected workflow instance from the respective execution server instances. Then, the adaptation agent adapts the control and data flow definition which is then further executed by the execution servers instances. In case temporal estimations are necessary, the adaptation agent requests duration information from the duration monitor. The workflow monitoring agent and the inter-workflow agent directly communicate with the execution servers instances as well, in order to monitor workflow instances respective to derive inter-workflow implications of adaptations.

For any operation that shall be performed on a workflow instance, the *authorization server* checks whether the requesting user or agent is authorized to request a particular workflow operation. To decide this it uses an authorization table. An excerpt of the authorization table for the HEMATO-WORK application is shown in Table 11-1. In this excerpt, it is specified that any staff member instance that fulfills the pattern in the topmost row (i.e., a staff member being a *senior* oncologist) is allowed to perform any global operation (i.e., start, abortion, or suspend) and any local operation

Staff Member Pattern (<i>Obj-Patt</i> < <i>Staff-Member</i> > according to 4.2.1.5)	Allowed operations
<i>Physician</i> [<i>degree</i> = <i>Senior</i> ; <i>speciality</i> = “ <i>Oncology</i> ”];	Global: All (start, suspend, abort)
	Local: All from Chapter 8, i.e., <i>cfop-drop-node</i> , <i>cfop-change-value</i> etc.
<i>Physician</i> [<i>degree</i> = <i>Assistant</i> ; <i>speciality</i> = “ <i>Oncology</i> ”];	Global: None
	Local: As for senior physicians

Table 11-1: Excerpt from authorization table for HEMATOWORK application.

such as dropping or adding activities. In contrast to this, an assistant oncologist is only allowed to perform local operations such as the adding or removing of single drug activities while the abortion or suspension of entire chemotherapies should be left up to an experienced senior oncologist.

More details about the workflow engine can be found in [DIETZSCH 2000].

11.3 Communication and Integration Layer

In this section we describe the middleware-based implementation of the AGENTWORK communication and integration layer. On the market, a broad range of middleware approaches exist, such as DCOM/COM+ [SESSIONS 1998, EDDON 1999], CORBA [BAKER 1997], ENTERPRISE JAVA BEANS [MONSON-HAEFEL 2000], and XML-based approaches such as BIZTALK [KOBIELUS 2000]. Among these, CORBA has been selected for our implementation of the communication and integration layer. This is because CORBA is more appropriate for large-scale environments than DCOM [THOMPSON & WATKINS 1997], and provides more appropriate services and infrastructure components for the specific purposes of control flow failure handling than for instance ENTERPRISE JAVA BEANS or BIZTALK [FEILER 2000]. For example, CORBA provides a so-called *event service* that allows to register and propagate events. This is of great use for the event monitoring agent which has to detect events that constitute control flow failures of running workflows (see 11.4.1). From the available CORBA implementations, IONA ORBIX (version 3.3) [IONA 2002] has been selected, as this product offers one of the most comprehensive CORBA implementations.

This section is organized as follows: First, we sketch the principal structure of CORBA in 11.3.1. In 11.3.2, we describe the CORBA-based implementation approach of the communication and integration layer. Concerning the connection to application databases and programs, we will use examples from the HEMATOWORK project. Note that our implementation approach is a straightforward one as the research focus of this thesis is on semi-automated control flow failure handling rather than on interoperability aspects. For enhanced middleware-based workflow system implementations considering additional aspects such as performance we refer to [DOGAC ET AL. 1998, SCHULZE 1999].

11.3.1 Principal Structure of CORBA

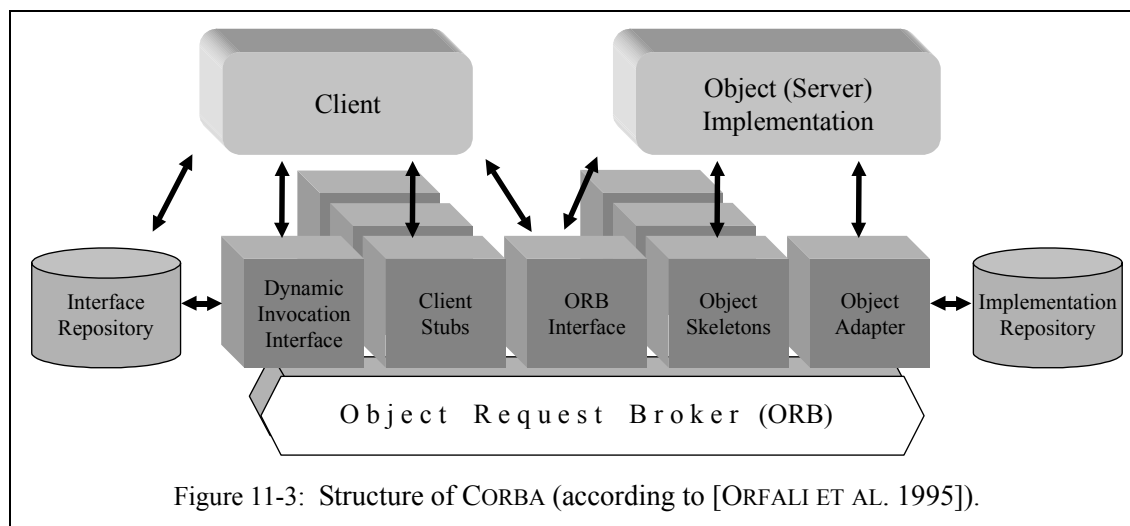
CORBA (Figure 11-3) has been designed to support interoperability between objects in a heterogeneous and distributed environment. It is based on the OMG object model [SOLEY & KENT 1995] which defines a common meta model for specifying the externally visible characteristics of objects in a standardized and implementation-independent way. In this model, clients request services from objects (which will also be called servers) through a well-defined interface. This interface is specified in the so-called *interface definition language* (IDL), e.g.,

```
interface example_server_object{  
    attribute    string s;  
    void         my_operation(in long a, out long b, inout boolean b);  
}.
```

This specification defines a server object which has a string attribute called *s* and on which a client can invoke the operation (or method) *my_operation* with three parameters (the *in/out/inout* entries specify whether the parameter value will be passed from client to server respective from server to client respective in both directions).

An IDL specification is then processed by an IDL compiler which generates client stubs and server skeletons in an implementation language such as C++ or JAVA, which are then included in the client's respective the server's program. IDL specifications can be stored in the so-called *interface repository*, so that the interface structure of objects can be inspected at execution time. To the server skeletons, application-specific code has to be added so that the server object can provide the necessary functionality.

A client accesses an object by issuing a *request* to this object. The request consists of the operation to be performed, the object reference, and actual parameters (if any). The object reference is a log-



ical object name that identifies an object reliably. For this, CORBA provides a naming service that allows to assign system-wide logical names to objects.

The central component of CORBA is the *object request broker* (ORB) which provides the communication infrastructure. The basic ORB functionality consists of passing the requests from clients to the server objects. In order to make a request the client can communicate with the ORB through the client stub (generated by the IDL compiler) or through the *dynamic invocation interface*. The latter allows the client to specify requests to objects whose definition and interface are unknown at the client's compile time. In order to use the dynamic invocation interface, the client has to dynamically compose a request. The needed information about objects and their attributes and operations is retrieved from the interface repository.

The communication between the object implementation and the ORB core is handled by the *object adapter*. It provides services such as the registration of implementations, implementation activation and deactivation, generation and interpretation of object references, and operation invocation. There exist many different special-purpose object adapters to fulfill the needs of specific systems such as databases [SELLENTIN 1999]. The information the object adapter needs for its tasks, such as an object's location and the operating environment, is stored in the so-called *implementation repository*.

The main advantage of the CORBA approach is first that programming is significantly facilitated as CORBA provides a lot of services which cover general-purpose functionality, such as the object request broker or the naming service mentioned above. In particular, client programming does not have to make any assumptions about the location of server objects, their internal organization, the operating systems on which they run, and many other details. Second, extensibility is very good, as new objects can easily be plugged into the system [ORFALI ET AL. 1995]. The main disadvantage of CORBA, i.e., the limited performance [SELLENTIN 1999], does not play such a central role for the addressed application classes as it would be the case for “real-time” applications such as intensive medicine. For example, in the HEMATOWORK application it is not necessary for most patient data that they are communicated within the next few seconds or minutes. Rather, it is sufficient if they are reported within the next one or two hours.

For a more detailed discussion of CORBA we refer to [SELLENTIN 1999, SCHULZE 1999]. Furthermore, when we say CORBA/C++ objects in the following, we mean objects that are defined in IDL, registered and controlled by CORBA, and implemented in C++.

11.3.2 CORBA-based Implementation of Communication and Integration Layer

We now describe our CORBA implementation approach of the AGENTOWORK communication and integration layer by using examples from HEMATOWORK. The implementation is based on the following principles (Figure 11-4): First ACTIVETFL classes are mapped to IDL interfaces. Second, in the implementations assigned to these interfaces, the connections to the HEMATOWORK data sources and application programs are encoded in C++. Third, at execution time, any access to an ACTIVETFL object is then translated to an access of the corresponding CORBA/C++ object.

1. Mapping from ACTIVETFL to IDL

For every ACTIVETFL class defined by the workflow editor, a corresponding IDL interface is generated providing the same attributes, relationships and methods. For example, for the ACTIVETFL class

Hemato-Finding[parameter: Enum{Leukocyte-Count, ...}, value: Float, unit: Enum{#/mm³, mg/mm³, ...}],
Hemato-Finding IS-A Laboratory-Finding (i)

the corresponding IDL interface

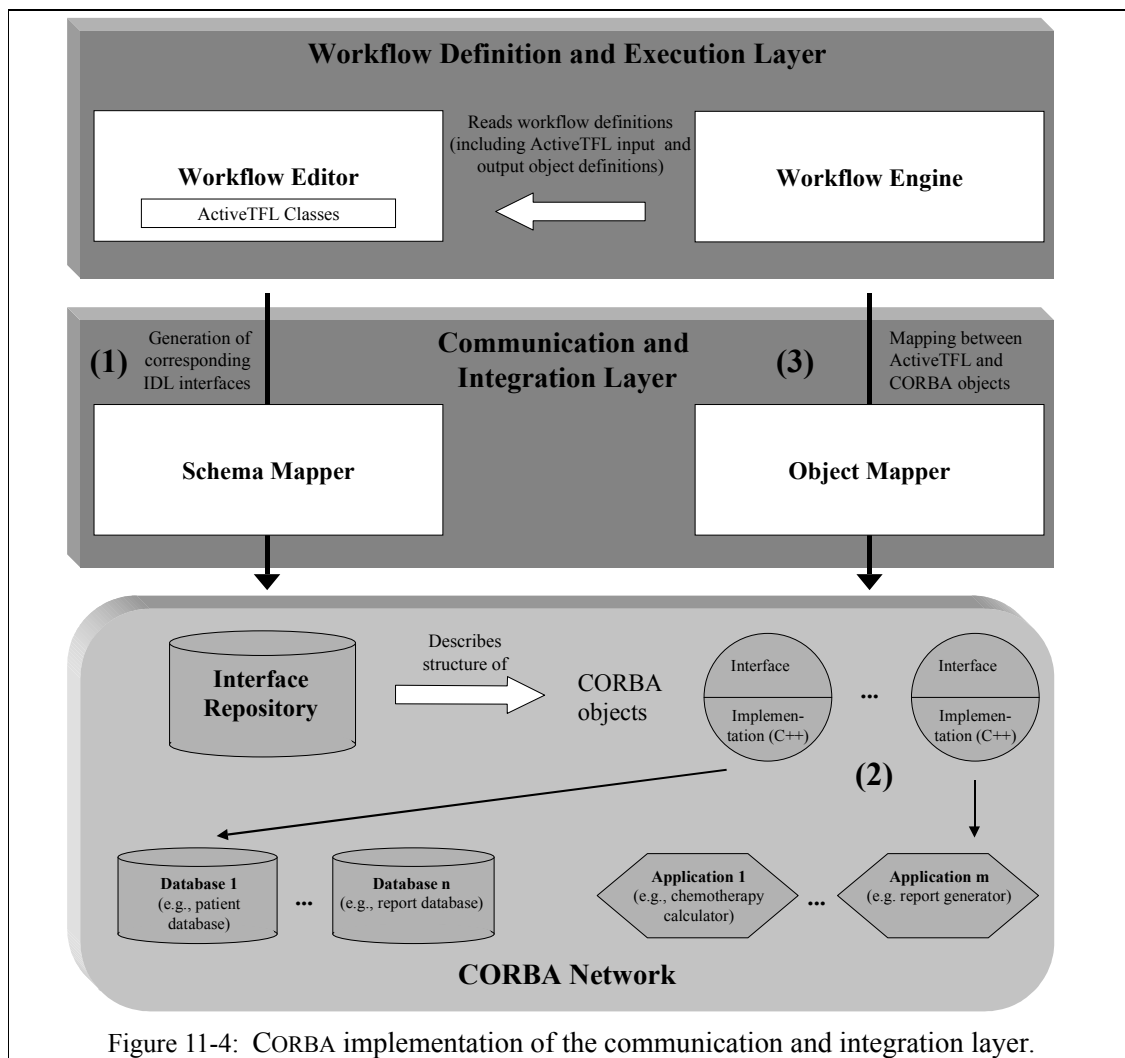


Figure 11-4: CORBA implementation of the communication and integration layer.

```

interface Hemato-Finding : Laboratory-Finding {
    enum paraType {Leukocyte-Count, ...};
    enum unitType {#/mm3, mg/mm3, ...};
    attribute paraType parameter;
    attribute float value;
    attribute unitType unit;}

```

(ii)

is generated (number **(1)** in Figure 11-4) and stored in the CORBA interface repository. The task of generating these IDL interfaces and of maintaining the relationships between the ACTIVETFL classes and their IDL counterparts is performed by the so-called *Schema Mapper*. The generated IDL interfaces are then processed by the IDL compiler which produces the client stub code and the object skeleton code. Note that for an attribute a reading operation is automatically generated for the stub by the IDL compiler, so that it does not have to be declared in the interface. For example, to read the value of the attribute *unit* declared in (ii), the generated stub contains the code

```

class Hemato-Finding: public virtual Laboratory-Finding {
    public: virtual float value() ...
}

```

(iii)

2. Connections to HEMATOWORK Data Sources and Application Programs

Within the C++ implementation assigned to an interface, it is encoded which database or application program has to be accessed when an object operation is invoked (number **(2)** in Figure 11-4). For example, in HEMATOWORK the C++ implementation assigned to the IDL interface in (ii) contains embedded SQL statements accessing the ORACLE-based patient database when the attribute values of a *Hemato-Finding* object have to be retrieved (by the respective reading operations). The connection between the interfaces and the implementation is handled by the CORBA object request broker as described in 11.3.1.

3. Mapping of ACTIVETFL Objects to CORBA/C++ Objects

At workflow execution time, the mapping between the ACTIVETFL object specifications of the workflow definitions and the CORBA objects is performed by the so-called *object mapper* (number **(3)** in Figure 11-4). For example, if the engine detects in the workflow definition that an object of the ACTIVETFL class *Hemato-Finding* as defined in (i) is needed as input object for an activity execution, it instructs the object mapper to retrieve the corresponding CORBA/C++ object. The object mapper then inspects whether the respective object already exists (as it for example has already been generated as an output object of a preceding activity execution), or generates a new one. These CORBA/C++ objects are then used further during the activity execution (e.g., sent to the application program supporting the activity execution). Furthermore, the object mapper manages every operation invocation on these CORBA/C++ objects.

Details about this “*Mapping to CORBA*” approach are described in [NEUBERT 1999]³.

11.4 Layer for Handling Control Flow Failures

We now describe the implementation of the layer for handling control flow failures. For this, we concentrate on the event monitoring agent (11.4.1) and the adaptation agent (11.4.2), as these two agents are the most complex ones. This is because the event monitoring agent closely has to interact with the entire workflow environment to register all relevant events, and as the adaptation agent has to estimate and to adapt running workflow instances. In particular, due to the complexity and importance of the adaptation agent, a simulation environment has been implemented for this agent to perform at least a minimal evaluation. The other agents such as the control agent and the workflow monitoring agent are not as complex as that and have been implemented in a straightforward manner, so that we omit details about their implementation. All agents have been implemented mostly in MS VISUAL C++.

11.4.1 Event Monitoring Agent

As described in Chapter 3 (*AgentWork Overview*), the task of the event monitoring agent is to decide which application events occurring somewhere in the AGENTWORK environment raise control actions. The main implementation decision in this context has been that all applications events are stored and processed *centrally* by this agent, with the consequence that all events first have to be registered and collected from the local application components (i.e., the data sources and application programs) where they are generated. The alternative that already the local application components themselves derive whether an event constitutes a control flow failure (e.g., by database triggers) is not suitable: First, control over the critical process of deriving control actions would be scattered over the local application components. Second, software maintenance would be complicated. For example, one would have to cope with different database trigger formats in case of different databases. Furthermore, any change of the control action structures would have to be implemented for every event-generating application component.

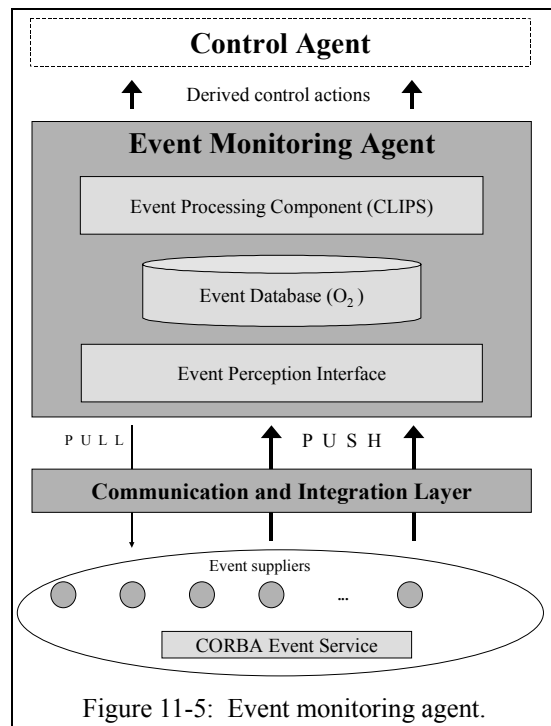


Figure 11-5: Event monitoring agent.

3. Though this master thesis describes a schema and object mapper not from ACTIVETFL to CORBA, but from the FLOWMARK DEFINITION LANGUAGE (FDL) of IBM FLOWMARK to CORBA, the implementation principles are the same (as only the source formats – FDL respective ACTIVETFL – are different).

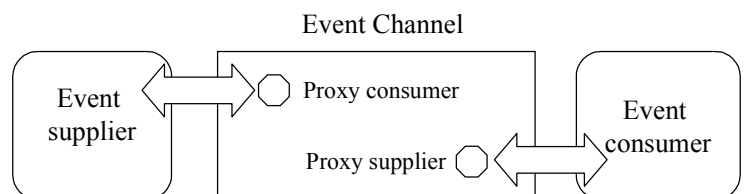
To achieve the task of a centralized event storage and processing, the event monitoring agent provides an event perception interface to receive events, an event database to store them, and an event processing component (Figure 11-5) to derive control actions which then are passed to the control agent for further processing. These components are now described in more detail.

11.4.1.1 Event Perception Interface

With the event perception interface, the event monitoring agent registers relevant events of the AGENTWORK environment, such as the event of inserting a new laboratory value into a patient database. The event perception interface has been implemented with ORBIXEVENTS [IONA 2002] which is the IONA implementation of the CORBA event service [OMG 2002 A]. ORBIXEVENTS provides the following functionality (Figure 11-6):

- By importing predefined ORBIXEVENTS modules, any CORBA object can become a so-called *event supplier* that generates and reports events. For example, in the HEMATOWORK application the most important event supplier is the CORBA object encapsulating the patient database. This is because many control flow failures in HEMATOWORK are raised by events in this database, such as by the insertion of new hematological data indicating a severe blood toxicity.
- Analogously, any CORBA object can play the role of a so-called *event consumer* that receives and processes events. For example, in AGENTWORK the event perception interface itself is encapsulated as a CORBA object that – in the HEMATOWORK application – “consumes” events of the patient database.
- Event communication is handled by so-called *event channels* which manage the transfer of events between suppliers and consumers. An event channel allows consumers to register interest in events of particular types, stores this registration information, accepts incoming events from suppliers, and forwards these events to registered consumers. In particular, an event channel provides so-called *proxy consumers* and *proxy suppliers* to which the event suppliers respective consumers connect to communicate their events and which hide the internal structure of the channel. Any number of suppliers can issue events to any number of consumers using a single event channel, and new suppliers and consumers can be easily plugged into the system. In addition, any supplier or consumer can connect to more than one event channel.
- For event transfer, it can be selected between the *push* or *pull model*. In the push model, the supplier initiates the transfer of events by sending events to consumers without request. In the pull model, the consumer initiates the transfer of events by requesting events from suppliers.

Figure 11-6: CORBA event service (ORBIXEVENTS).



For the event perception interface, the push model has been selected as default to achieve a high currentness of the event situation (assuming that every event supplier communicates its events directly⁴). The alternative pull model where the event perception interface actively requests events from the relevant event suppliers is principally inadequate as in this case important and time-critical control flow failures perhaps may be detected too late. The pull model is only used, if for a new event the past temporal context of this event has to be known as well, and if the events constituting this past context are not already stored in the event database.

The main advantage of using the CORBA event service is that communication partners do not have to make any assumptions about one another (such as about the number and internal structure of their partners), and that an infrastructure for the event transfer is provided.

Table 11-2 shows two simplified sample code fragments (e.g., without exception handling) of the event perception interface implementation. In this example, the insertion of new hematological data in the patient database **(A)** triggers the generation of a CORBA/C++ *Hemato-Finding* object representing this event and its pushing to the event perception interface:

- The supplier code fragment **(B)** contains code of the CORBA/C++ object encapsulating the patient database. During initialization, an event channel is opened (line 1), and a channel administration object (line 2) and a proxy consumer object (line 3) are obtained for the supplier. The operation *create_and_push_hemato-finding* (line 4) is invoked by a database trigger when an insert occurs on the table for the hematological data. This operation first creates and fills a *Hemato-Finding* object with the event data (line 5)⁵. Then, it pushes this object to the consumer proxy (line 6). Internally, the proxy then passes this object through the channel and the push operation is automatically invoked on all consumer objects that are interested in this event type.
- The consumer code fragment **(C)** contains code of the CORBA/C++ object encapsulating the event perception interface. During initialization, an event channel is opened (line), and a channel administration object (line) and a proxy supplier object (line) are obtained for the supplier. Event perception and further processing is done by the application-specific implementation of the *push* operation (line). First, the received event is stored persistently in the event database (line ; see 11.4.1.2 as well). Second, the event processing component is launched to see whether this new event constitutes any control flow failure (line ; see 11.4.1.3 as well).

11.4.1.2 Event Database

Events received by the event perception interface above are stored in the *event database*. To achieve a high autonomy and to reduce communication costs, this event database is a full replication of all received events. In particular, this is of advantage as the workflow-related implications of a new event often can only be determined in the temporal context of other, prior events (e.g., in

4. For example, for relational databases this can be achieved by triggers.

5. *patId* is a parameter used to initialize the *of* attribute which is declared in the *Event* super class of *Hemato-Finding*, and which refers to the case (i.e., patient) to whom the event occurred.

CREATE TRIGGER PUSH-NEW-HEMATO-EVENT
AFTER INSERT OR UPDATE
ON HEMATO-FINDINGS

CALL create_and_push_hemato-finding
(:new.PAT-ID, :new.PARAMETER,
:new.VALUE, :new.UNIT)

(A)

TABLE HEMATO-FINDINGS			
PAT-ID	PARAMETER	VALUE	UNIT
xrsd622	Thrombocyte-Count	60000	#/mm ³
gbfh922	Leukocyte-Count	900	#/mm ³
...

Patient Database (Oracle)

Supplier code fragment (B) ↓

```

CosEventChannelAdmin::EventChannel_var ecVar;           // Declarations for:
CosEventChannelAdmin::SupplierAdmin_var saVar;          // Event channel
CosEventChannelAdmin::ProxyPushConsumer_var ppcVar;     // Event channel administration
CosEventChannelAdmin::ProxyPushConsumer_var ppcVar;     // Consumer proxy

/***** Initialization *****/

1. ecVar = EventChannel::_bind ("hemato-events", "medHost"); // Get event channel reference
2. saVar = ecVar->for_suppliers ();                          // Get supplier admin. object
3. ppcVar = saVar->obtain_push_consumer ();                  // Get proxy push consumer

/***** Event generation and event pushing *****/

4. void create_and_push_hemato-finding                     // Operation that creates new
   (short patId, paraType parameter, float value, unitType unit { // Hemato-Finding object and pushes
       Hemato-Finding *e;                                         // it through the event channel

5.     e = new Hemato-Finding(patId, parameter, value, unit); // Create Hemato-Finding object
6.     ppcVar->push (*e); ...}                                   // Push it to event channel

```

Consumer code fragment (C) ↓

```

CosEventChannelAdmin::EventChannel_var ecVar;           // Declarations for:
CosEventChannelAdmin::ConsumerAdmin_var caVar;          // Event channel
CosEventChannelAdmin::ProxyPushSupplier_var ppsVar;     // Event channel administration
CosEventChannelAdmin::ProxyPushSupplier_var ppsVar;     // Supplier proxy

/***** Initialization *****/

ecVar = EventChannel::_bind ("hemato-events", "medHost"); // Get event channel reference
caVar = ecVar->for_consumers ();                          // Get consumer admin. object
ppsVar = caVar->obtain_push_supplier ();                  // Get proxy push supplier

/***** Event perception and further processing *****/

virtual void push (Event& e) {                             // Consumer implementation of push
    store-in-event-database(e);                           // Store new event in database
    launch-event-processing-component();...}               // Check for control flow failures

```

Table 11-2: Implementation of event perception interface.

case of time series events). Without the replication of the whole event history, the event monitoring agent often would have to request the original physical data sources when reasoning about events. If such a source would not be available at the moment of the request, event processing could be delayed with the consequence that perhaps important control flow failures would be detected too late. Event updates in the original physical data sources can also be specified as relevant events inducing event pushing so that a high currentness of the replicated event database can be achieved.

Depending on the application, events being older than an application-specific temporal threshold may be deleted from the event database if it can be assumed that they are not needed anymore. For example, in HEMATOWORK laboratory events older than 6 months can be dropped from the event database of the event monitoring agent as they usually are not relevant anymore for control flow failure handling. Nevertheless, in HEMATOWORK they still remain in the patient database because of legal requirements, so that they always can be “pulled into” the event database again.

For the implementation of the event database, the object-oriented database system O₂ [DEUX 1991] (version 5.0.2) has been used as it supports the ODMG⁶ object model [CATTELL ET AL. 2000] which itself fits well into the CORBA/C++ object model. In addition, the O₂CORBA component [O2 1998] supports the connection of an O₂ database to CORBA by providing, for instance, the automatic generation of IDL interfaces from an O₂ database schema.

11.4.1.3 Event Processing Component

The task of this component is to process events on the basis of the ACTIVETFL failure rules as they have been introduced in Chapter 4. The ACTIVETFL failure rules have been implemented with the rule-based programming environment CLIPS⁷ (version 6.1) [GIARRATANO & RILEY 1993] (while VISUAL C++ is used for the higher-level procedures). CLIPS has been selected because of the following reasons:

1. Support of Forward Chaining Rule Processing

In contrast to other rule-based programming environments such as PROLOG, CLIPS supports a *forward chaining* (or *data-driven*) rule processing mode [BUCHANAN & SHORTLIFFE 1984]. This processing mode means that whenever new data becomes available it is checked for every available rule whether the WHEN/WITH part of the rule becomes true. If this is the case, the rule is triggered and new data (such as a control action) is generated by the THEN part. In contrast to this, PROLOG and most other rule-based programming environments support *backward chaining* (or *hypothesis-driven*) rule processing. This processing mode means that first a hypothesis is generated (such as the hypothesis that some control action holds for some activity pattern and case). Second, it is checked whether there is any data and rule constellation that can verify this hypothesis [SCHÖNING 1989].

Forward chaining is appropriate for so-called monitoring problems [SHAHAR & MUSEN 1996, MOSTERMAN & BISWAS 1997, LARSSON & HAYES-ROTH 1998] where large amounts of data

6. Object Data Management Group

7. C-Language Integrated Production System

continuously have to be scanned w.r.t. any possible implications, and where the number of such possible implications (i.e., hypotheses) is too large to work with backward chaining. In contrast to this, backward chaining can efficiently solve problems where the number of relevant hypotheses can be limited (e.g., by user input), and where the amount of data is not too large. As control flow failure handling can be viewed as a monitoring problem – a lot of events have to be monitored whether they constitute any control flow failures – forward chaining is the more appropriate processing mode for AGENTWORK.

2. *Object-Oriented Data Model*

CLIPS supports a full object-oriented data model through its part COOL (CLIPS Object-Oriented Language) which is compatible to C++. Thus, the CORBA/C++ event objects that are registered and maintained by the event monitoring agent can easily be mapped to CLIPS objects which are then processed by CLIPS rules.

3. *Powerful C/C++ APIs*

Furthermore, CLIPS provides a powerful C/C++ API. Thus, for the high-level procedures of the event monitoring agent which have to control rule processing and which are written in C++, the availability of such a C/C++ API is of great usage. For example, by this API it is possible during rule processing to add routines for user interaction or to handle dynamic dependencies between control actions according to 7.6.

Table 11-2 shows the CLIPS notation of the *Hemato-Finding* class of 11.3.2 and a simplified CLIPS rule deriving a *drop* control action when a leukocyte count is less than 1000 #/mm³. Rules such as the one shown in Table 11-2 operate on the CORBA/C++ event objects received by the event perception interface and process them in the above-mentioned forward chaining mode. The control actions derived by such rules are then passed to the control agent, which performs workflow abortions or suspensions in case of global control actions, or instructs the adaptation agent to adapt a workflow dynamically in case of local control actions.

11.4.2 *Adaptation Agent*

In this section we describe implementation issues concerning the adaptation agent. First, we describe the internal structure of the adaptation agent (11.4.2.1). Second, we describe how the adaptation agent internally marks a workflow to indicate for which area which control actions hold and which adaptation strategy is performed for this area (11.4.2.2). Third, we describe a simulation environment to support the evaluation of workflow adaptations (11.4.2.3).

11.4.2.1 *Internal Structure*

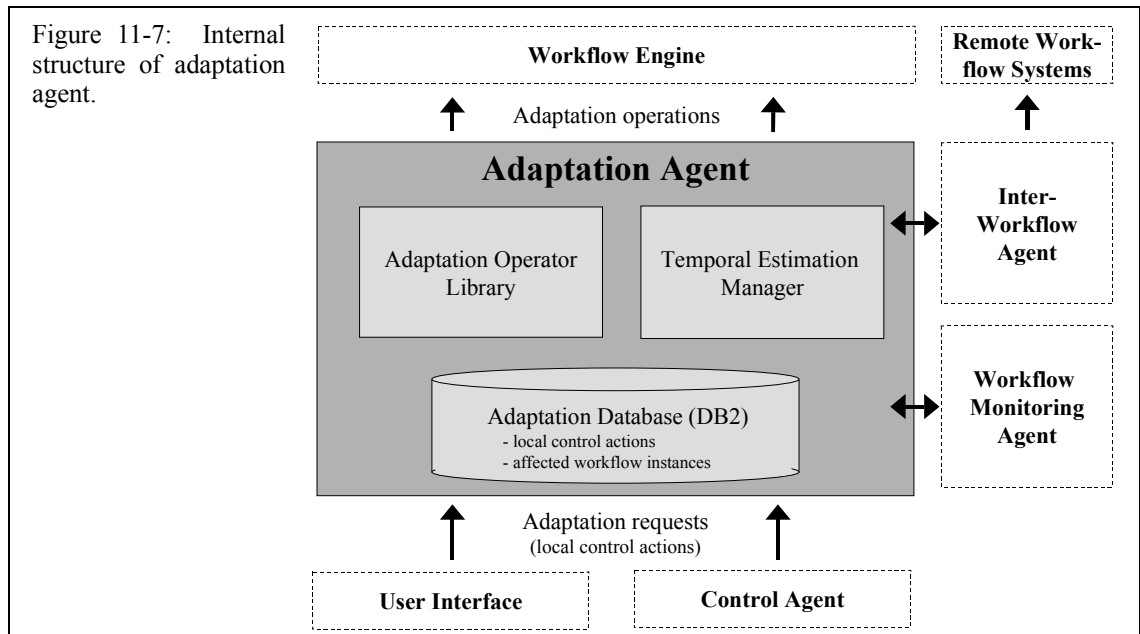
The adaptation agent is organized as follows (Figure 11-7): A DB2-based *adaptation database* stores all adaptation-related information, such as the local control actions that have been passed on by the user interface or the control agent. The *temporal estimation manager* performs the duration estimations specified in Chapter 6. It has been implemented as an own submodule as its functional-

CLIPS examples		Comments
Notation of class <i>Hemato-Finding</i>	<pre>(defclass Hemato-Finding (is-a Laboratory-Finding) (slot parameter (type SYMBOL) (allowed-values Leukocyte-Count, ...)) (slot value (type FLOAT)) (slot unit (type SYMBOL) (allowed-values #/mm³, mg/mm³, ...)))</pre>	defclass = define class slot = attribute SYMBOL = any sequence of printable ASCII characters (used for enumeration types)
Class to represent drop control actions	<pre>(defclass Drop-Control-Action (is-a Control-Action) (slot activity-to-be-dropped (type Activity)) (slot valid-time-start (type DATE)) (slot valid-time-end (type DATE)))</pre>	Simplified CLIPS class to represent <i>drop</i> control actions (only for fixed valid time). <i>Control-Action</i> is a superclass storing the case for which the control action holds. <i>Activity</i> is a CLIPS base class corresponding to the ACTIVETFL base class <i>Activity</i> (introduced in 4.2.1.1)
Rule generating a control action	<pre>(defrule severe-hemato-status (object (is-a Hemato-Finding) (parameter Leukocyte-Count) (value ?v&:(< ?v 1000)) (unit #/mm³)) => (make-instance of Drop-Control-Action (activity-to-be-dropped (make-instance of Drug-Administration (drug "Etoposid"))) (valid-time-start now) (valid-time-end 7/8/2001)))</pre>	defrule = define rule Defines a rule with the following meaning: If there is a leukocyte count being less than 1000 #/mm ³ , then generate instance of class <i>Drop-Control-Action</i> which has as a subcomponent an instance of class <i>Drug-Administration</i> (subclass of <i>Activity</i>) where slot <i>drug</i> is set to the string " <i>Etoposid</i> "

Table 11-2: CLIPS examples.

ity is frequently used by the inter-workflow agent as well, which has to compare the durations of a workflow before and after an adaptation. The *adaptation operator library* implements the structural adaptation operators that have been introduced in Chapter 8.

Figure 11-7: Internal structure of adaptation agent.



11.4.2.2 Internal Representation of Adaptation Areas

On the conceptual level described in the chapters before, we have restricted our considerations to a few important adaptation cases such as that all control actions holding simultaneously for a workflow have the same valid time interval. This has been reasonable as the general case could have been derived from these special cases. However, on the implementation level we have to consider that different control actions with different valid time intervals hold simultaneously for the same workflow and that they furthermore are handled by different adaptation strategies. Thus, it has to be stored precisely which parts of the workflow are affected by which control action and handled by which adaptation strategy is not only important for the adaptation agent itself, but for the workflow monitoring and the inter-workflow agent.

To cope with this, the adaptation agent inserts so-called *adaptation area* nodes to mark those parts of the workflow handled reactively or predictively for a control action. Such adaptation area nodes form a further type of control nodes with an execution duration being zero. When the workflow engine detects one of them, it simply ignores it and continues with the execution of the successor nodes. Thus, the execution model of Chapter 5 does not have to be extended. We distinguish two types of adaptation area nodes (Figure 11-8):

- So-called PRED-END nodes mark the end of the workflow part that has been predictively adapted due to some control action being valid during the interval VT . The information for which control action(s) such a PRED-END node holds and when it should be reached according to the estimation is stored in a table ADAPTATION-AREAS in the adaptation database. For

example, in Figure 11-8 it is stored that PRED-END nodes 5 and 9 close the workflow part that has been adapted because of a *drop(A, C)* control action, and that according to the estimation these nodes should be reached on 20th September 2001 at 8 pm as then the valid time interval terminates. Such a termination calendar point in time is derived by adding the duration of the control action's valid time to the point in time when the control action has been triggered.

Note that nodes to mark the beginning of an adaptation area are *not* necessary. This is because due to the valid time conventions for control actions introduced in 7.3, a valid time interval always starts at the point in time when the control action has been triggered. Thus, it cannot occur that a not yet executed predecessor node of the PRED-END node does *not* belong to the adaptation area closed by the PRED-END node.

- So-called REACTIVE-START nodes mark the beginning of areas that could not have been estimated by the adaptation agent and thus have to be handled reactively. In the example of

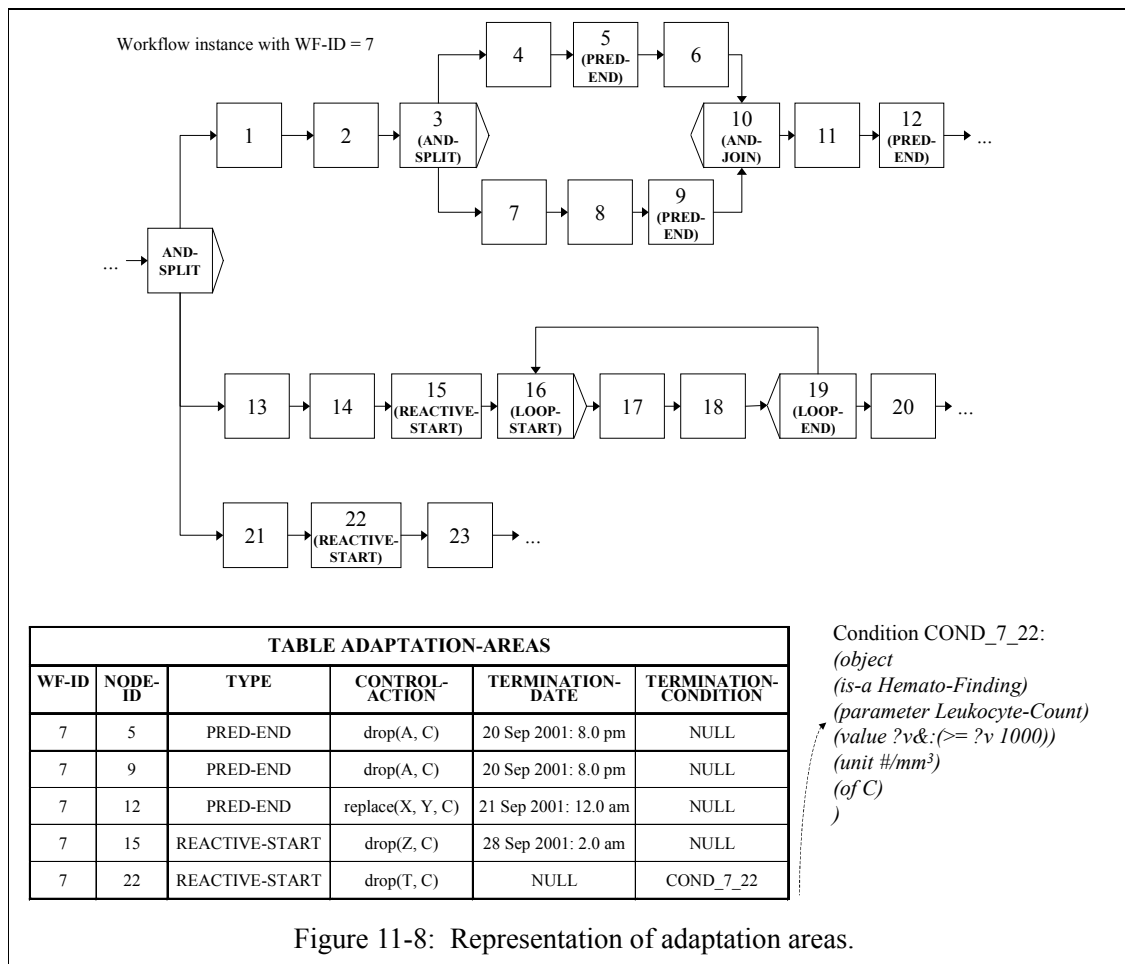


Figure 11-8, it is shown that from REACTIVE-START node 15 on the workflow has to be adapted reactively (as the loop could not have been estimated). REACTIVE-START node 22 marks the beginning of an adaptation part that cannot be estimated not because of missing temporal information but because of a conditional end of the associated valid time interval. For this node, the respective condition in CLIPS notation stating that the leukocyte count has to be at least 1000 #/mm³ is shown in the TERMINATION-CONDITION entry in the last row in the table of Figure 11-8 (again, the *of* slot in the *Hemato-Finding* object is inherited from a super-class *Event* and refers to the case for whom this condition has to hold).

Note that nodes to mark the *end* of the area to which reactive adaptation is applied make no sense, as this end is not known (otherwise reactive adaptation would not have been selected).

Note that the location of different PRED-END and REACTIVE-START nodes may not be as “path-balanced” as shown in Figure 11-8. For example, node 2 in the upper path could be a REACTIVE-START node as well, indicating that from this node on the workflow has to be adapted reactively for some control action different from the control actions assigned to the PRED-END nodes 5, 9, and 12.

The information stored in the table ADAPTATION-AREAS is then directly used by the workflow monitoring to check whether the estimations of the adaptation agent meet the execution reality.

11.4.2.3 Simulation Environment

The adaptation of a workflow instance and especially its *predictive* adaptation has to be viewed as a complex task so that an evaluation of the temporal estimation algorithms and the operators adapting a workflow instance is necessary. However, a real-world evaluation such as one in a medical environment has not been possible due to the incomplete implementation of the HEMATOWORK application. Therefore, a simulation environment has been implemented to achieve at least some minimal evaluation. This simulation environment provides the following functionality:

1. Generation of Workflow Definitions

At random, arbitrary workflow definitions can be generated on the basis of the block-oriented workflow definition model described in 5.3. To avoid that unrealistic workflows are generated, the maximal number of nodes and loops and the maximal nesting order for blocks can be specified. In the performed evaluations, the maximal number of nodes, loops and block nestings has been restricted to 50 nodes respective 2 loops respective 3 block nestings per each workflow.

2. Generation of Local Control Actions

An arbitrary number of local control actions including their valid time intervals can be generated at random to simulate a control flow failure. For this, the maximal number of simultaneously generated local control actions can be specified (e.g., 4 in the performed evaluations).

3. Generation of Workflow Instances

To simulate a running workflow instance being affected by the generated control actions, execution states are assigned at random to the nodes and edges on the basis of the workflow execu-

tion model described in 5.4. In particular, a failure node set (7.4.1) is selected at random to simulate the execution focus (i.e., nodes that are currently executed or will be executed next) of the workflow instance at the point in time of control action triggering.

These simulated workflow instances are then estimated and adapted by the adaptation agent. As the data environment is not simulated in the current version, branching conditions at OR-SPLIT or LOOP-END cannot be resolved predictively. To compensate this, the conditional paths that are executed respective the number of loop iterations are determined at random as well.

On the basis of this simulation environment, a representative number (about 50) of generated workflows has been adapted. The adapted workflows have been inspected manually to check whether the adaptation agent has worked properly for these test workflows. More details about the adaptation agent can be found in [GREINER 2000].

11.5 Summary

In this chapter, we have described the implementation of the AGENTWORK prototype. First, we have sketched the implementation of an adaptation-oriented workflow management system that allows to define and execute ACTIVETFL-based workflows. In particular, it has been described that workflows are executed as interpreted structures by a multi-threaded workflow engine. Second, we described the implementation approach of the communication and integration layer. For the purposes of distribution and heterogeneity transparency and scalability, this layer has been implemented by encapsulating the data sources and application programs of the AGENTWORK environment within CORBA/C++ objects, and by translating all data and program requests generated at workflow execution time to operator invocations on such CORBA/C++ objects. Third, we described implementation issues of the layer for handling control flow failures. In particular, we described how the CORBA event service has been used to implement the registration and transfer of events, and how failure rules deriving control actions have been implemented with the rule language CLIPS. Furthermore, we described the simulation environment of the adaptation agent that supports the evaluation of workflow adaptations.

In the future, the described implementation has to be completed. For example, the current evaluation of generated test workflows in the described simulation environment can be only viewed as a first step of evaluation. One limitation of the simulation environment is that it does not simulate the data flow and the continuation of a workflow after the adaptation. Thus, the predictive resolving of conditions and workflow monitoring could not have been tested so far. Furthermore, the implementation of the workflow definition and execution layer so far has neglected components not absolutely necessary (such as the worklist handler), and therefore has to be enhanced. Recent encouraging results of our database group concerning the usage of the ADEPT_{FLEX} workflow management system [REICHERT 2000, HENSINGER ET AL. 2000] have been obtained after the completion of the AGENTWORK implementation described in this thesis, so that they could not have been considered anymore. The possibility to use the ADEPT_{FLEX} system for a new version of AGENTWORK is discussed in Chapter 12 (*Summary and Future Work*).

The overall goal of this thesis has been to support the semi-automated handling of so-called control flow failures, and thus to make workflow management more flexible w.r.t. application events. After we have described our failure handling approach in detail, this chapter completes the thesis by summarizing the results in Section 12.1, and by describing future work in Section 12.2.

12.1 Summary

In this section, we summarize the contributions of AGENTWORK to adaptive workflow management. Table 12-1 repeats the requirements that have been identified for adaptive workflow management systems in Chapter 2 (*Related Work*), and shows to what degree AGENTWORK fulfills them.

In Chapter 1 (*Introduction and Problem Description*), we have introduced control flow failures as an important failure type which is caused by application events and characterized by an inadequacy of control flow. We have motivated that for many workflow application classes such as medicine or insurance business the handling of this failure type cannot be neglected. In Chapter 2 (*Related Work*), we showed that existing approaches from the fields of commercial workflow management systems, advanced transaction models, exception handling in programming languages, adaptive and collaborative workflow management, and artificial intelligence do not support the automated handling of control flow failures sufficiently. In particular, we have seen that especially the temporal structure of control actions (requirement 1.3), the support of predictive adaptation (requirement 2.3), and the handling of inter-workflow implications of control flow failures (requirement 3) (see Table 12-1) are not supported sufficiently by current approaches. The main motivation for consid-

Central Requirements		Subrequirements	Support	Supporting Approaches and Remarks
1.	Representation of Failure Events and Control Actions	1.1 High Semantic Level of Event and Control Action Representation	Yes	ECA rules for control flow failures on the basis of temporal object-oriented logic ACTIVETFL (Chapter 4 and Chapter 7)
		1.2 Temporal Structure of Events	Yes	
		1.3 Temporal Structure of Control Actions	Yes	
		1.4 Integrity of Failure Rules	Yes	
		1.5 Authorization of Control Actions	Yes	
2.	Translation of Control Actions into Workflow Execution Operations and Structural Adaptations	2.1 Workflow Abortion and Suspension	Yes	Control and data flow constraints, state-based workflow execution model (Chapter 5) Algorithms for workflow estimation (Chapter 6) Library of control and data flow adaptation operators (Chapter 8) Control action ordering in case of simultaneous control actions, workflow monitoring (Chapter 9)
		2.2 Support of Reactive Adaptation	Yes	
		2.3 Support of Predictive Adaptation	Yes	
		2.4 Consideration of Data Flow Implications	Yes (partially)	
		2.5 Consistency of Adapted Workflows	Yes (partially)	
		2.6 Efficiency of Adaptation	(No)	Not specifically supported, but negligible as real-time applications are not addressed by AGENTWORK.
3.	Handling of Inter-Workflow Implications of Control Flow Failures	3.1 Determination of Temporal Implications	Yes	Temporal and qualitative constraints between cooperation partners Workflow estimations and application of quality transformation rules to check whether agreed-on temporal and qualitative constraints are violated by control flow failure Report of constraint violations to cooperation partner (Chapter 10)
		3.2 Determination of Qualitative Implications	Yes	

Table 12-1: Support of requirements by AGENTWORK.

ering the temporal structure of control actions has been that such control actions often do not only hold for “a moment” or “for ever”, but for a clearly specified time such as the next seven days. The main motivation for the support of predictive adaptation and the handling of inter-workflow implications has been that workflow users and cooperation partners have more time to prepare themselves w.r.t adapted workflows. Thus, this thesis focussed especially on these requirements 1.3, 2.3, and 3.

To meet these requirements without neglecting the other ones, this thesis has provided the following approaches:

1. Logic-Based ECA Rules for Control Flow Failures (for central requirement 1 in Table 12-1)

To express which application events induce which actions for workflows (e.g., abort or suspend) or activities (e.g., drop or add), ECA rules for control flow failures have been introduced. These ECA rules are specified with ACTIVETFL, which is a temporal object-oriented logic that has been designed for the purposes of this thesis. It combines a high expressiveness with a formal basement. The ECA rules allow us to specify control flow failures on a **high semantic level** meaningful for users such as physicians, without making any assumptions about the physical representation of events and actions (requirement 1.1). In particular, on the event-side, these rules allow not only to express single events such as the insertion of a new laboratory value into a database, but also composite events with a complex **temporal structure** such as time series events (requirement 1.2). On the action side, global and local control actions can be specified and provided with fixed or conditional valid times, to specify for which time frame they hold (requirement 1.3). Furthermore, this thesis has provided mechanisms to achieve the **integrity of failure rules** in terms of rule redundancy, rule incompatibility, and rule termination (requirement 1.4). Additionally, an **authorization concept** on the basis of staff member patterns has been implemented to restrict the application of control actions to staff members that are qualified for this task (requirement 1.5).

Summarizing, the ECA failure rules provided by this thesis allow us to cover a broad range of control flow failures. This has been shown at least for the medical case. As medicine can be viewed as one of the most complex application classes, it can be assumed that the provided rule approach is suitable for many other application classes as well.

2. Strategies and Operators for Workflow Adaptations (for central requirement 2 in Table 12-1)

For the translation of control actions into workflow execution operations and structural adaptations, several strategies and operator libraries have been implemented. In case of global control actions, workflow **abortion** or **suspension** can be performed (requirement 2.1). In case of local control actions, two different strategies are provided, namely *predictive* and *reactive* adaptation (requirements 2.2 and 2.3). The strategy of predictive adaptation is selected

- if a *fixed* valid time (such as for the next seven days) is assigned to the local control action so that AGENTWORK can estimate which workflow part will be executed during the valid time, and
- if it is known at the moment of the failure event w.r.t. which cases or resources those activity nodes will be executed that match the activity pattern of the control action.

If these conditions hold, the workflow part that is assumed to be executed during the assigned fixed valid time is adapted predictively by applying structural adaptation operators to it. The needed temporal estimations are performed on the basis of duration values that either have been specified at workflow definition time or have been obtained by execution time measurements. A particular strength of the estimation approach is that the duration of data flow processes such as requesting data from a user interface is considered, and that the duration values are grouped along several dimensions, such as the activity type, the staff member (group) or the application programs executing the activity, and the day time. Furthermore, if multiple control actions affect one workflow simultaneously, AGENTWORK determines a suitable order in which these control actions should be processed to minimize and control so-called pull-in and push-out effects. When a workflow has been adapted predictively, it is monitored after its continuation to check whether the estimations match the execution reality. If necessary, the adaptations are corrected.

The strategy of reactive adaptation is selected whenever the conditions for predictive adaptation are not met. For example, if a *conditional* valid time has been assigned to a control action it is impossible to predict for how long the control action will hold. Another reason may be that though a fixed valid time has been assigned, a temporal estimation is not possible because of unresolvable OR-SPLIT nodes or missing execution duration values. Reactive adaptation checks a node directly before it shall be executed, e.g., for a $drop(A, C)$ control action it is checked for every node n that is reached by the control flow during the valid time interval assigned to $drop(A, C)$, whether n is based on activity definition A and shall be executed for case C . If this is fulfilled, n is dropped from the control flow.

Both strategies use a library of adaptation operators that perform the necessary adaptations on the structural level by inserting or dropping single nodes, edges, or blocks.

For both adaptation strategies, the **data flow implications** are considered as well (requirement 2.4). For example, if a node is dropped that should have provided its output objects for one of its successor nodes, a new data flow edge is generated to compensate the missing of these output objects. This is done first by exploring the temporal neighborhood of the affected nodes or edges to check whether there are any existing nodes that could provide the needed output objects. Then, new internal data flow edges are generated that map these output objects to the nodes or edges needing them. If the local temporal neighborhood does not provide the needed objects, an F-Logic query is generated and assigned to an external data flow edge to retrieve the needed objects from an external data source. However, limitations of the described data flow approach include that an intensive user interaction may be needed (depending on the workflow structure), and that the complexity of generated F-Logic query expressions is limited, e.g., joins over different object extensions are not yet possible.

Furthermore, the **consistency** of adapted workflows is supported (requirement 2.5). This has been achieved by the introduction of several control and data flow constraints, such as that a conditional branching has to be closed properly by a corresponding joining node, or that the input of an activity node has to be completely provided by the data flow. By implementing the structural adaptation operators in a way enforcing these constraints, and by not allowing any adaptation not using these

operators, violations of these control and data flow constraints can be avoided. Furthermore, by clearly specifying the workflow execution model on the basis of edge and node execution states and by precisely stating for every adaptation operator how the execution states may have to be adapted for affected nodes, unclear and inconsistent execution semantics of adapted workflows are avoided.

For the efficiency of adaptations (requirement 2.6), no specific approach has been provided which however can be neglected as AGENTWORK does not address real-time applications where this would be a critical point.

Summarizing, the strategies and adaptation operators provided by this thesis allow to translate control actions into structural adaptations on the edge and node level in a way preserving the control action's semantics and the consistency of the workflow. In particular, whenever possible predictive adaptation is selected as this gives the staff more opportunity to prepare itself w.r.t. new situations.

3. Mechanisms for Handling Inter-Workflow Implications of Control Flow Failures (for central requirement 3 in Table 12-1)

Furthermore, an approach to deal with control flow failures for cooperating workflows has been introduced. This approach allows to determine the **temporal** and **qualitative implications** a workflow abortion, suspension, or dynamic adaptation may have for cooperation partners (requirements 3.1 and 3.2). Temporal and qualitative constraints can be assigned to communication nodes so that cooperation partners can specify in which time frame and quality range results should be provided. If a workflow is aborted, suspended or dynamically adapted, it is checked by workflow estimations and the application of quality transformation rules whether agreed-on temporal and qualitative constraints are violated. If this is the case, such constraint violations are immediately communicated to affected cooperating workflow systems. The affected cooperation partner then can handle such a constraint violation manually or by failure handling rules stating how to abort, suspend, or adapt its own workflows to cope with the new situation. By this approach, the frequency of failure situations inducing a workflow abortion, suspension, or dynamic adaptation but *not* reported timely to affected cooperation partners can be reduced.

4. Prototypical Implementation

Finally, a prototypical implementation of the AGENTWORK system has been described. This implementation consists of an adaptation-oriented workflow management system that allows to define ACTIVETFL-based workflows and to execute, abort, or suspend them. The internal representation of workflow instances supports their dynamic adaptation. As its main contribution, the implementation consists of agent prototypes of the layer for handling control flow failures. In particular, an implementation of the event monitoring agent is provided which uses the CORBA event service to register events and which derives control actions from these events by CLIPS rules. Furthermore, a simulation environment of the adaptation agent to evaluate workflow adaptations is provided.

Additionally, the implementation supports the integration into distributed and heterogeneous environments by encapsulating the data sources and application programs of the AGENTWORK environ-

ment within CORBA/C++ objects, and by translating all data and program requests generated at workflow execution time to operator invocations on such CORBA/C++ objects.

12.2 Future Work

In the future, we plan to work on the following topics:

1. Implementation

An encouraging perspective is to use the ADEPT_{FLEX} workflow management system [REICHERT 2000, HENSINGER ET AL. 2000] as core system for AGENTWORK. The ADEPT_{FLEX} system has kindly been provided by the database section (Head: Prof. Dr. Peter Dadam) of the Department of Computer Science, University of Ulm, Germany, to check whether it can be integrated into the AGENTWORK system.

From the AGENTWORK perspective, the key advantage of ADEPT_{FLEX} is that it provides a comprehensive set of adaptation operators (as described in 2.4.1) *and* that it provides a JAVA API to access these adaptation operators by external programs. Thus, we will investigate

- first whether we can replace the current AGENTWORK workflow definition and execution layer by that of ADEPT_{FLEX}, and
- second whether the adaptation agent can use the ADEPT_{FLEX} adaptation operators to adapt workflow instances.

Additionally, we will investigate whether other ADEPT_{FLEX} components may be usable for AGENTWORK as well, such as its powerful *organization modeler* or its *distribution layer* which supports the migration of workflow instances between different workflow engines.

2. Additional Control Actions

In addition to the control actions introduced in this thesis, we want to investigate whether there are further useful control actions that should be implemented, such as control actions that are able to adapt not only activity nodes but branching conditions as well. For example, let us assume a cancer patient who is very predisposed for infections. Let us furthermore assume that a workflow definition contains the conditional branching that some immunosuppressive cytostatic drug is only given if the leukocyte count is higher than 1000 (otherwise, an alternative, less immunosuppressive drug is administered). Then, for this infection patient it may be suitable to adapt this conditional branching in the sense that the immunosuppressive drug is only applied when the leukocyte count is higher than 1500 (instead of 1000), to avoid that his immune system is burdened too much.

3. Usage of Failure Rules to Construct Workflows

Another topic to investigate is not only to adapt existing workflows, but to dynamically *construct entire workflows* to deal properly with unexpected situations. For example, in medical disciplines such as intensive medicine some alert situations such as shocks require very patient-specific treatment workflows which cannot be predefined as the order in which activities have to be executed


```

WHEN
INSERT ON clinical-findings
WITH      new.type = "Anaphylactic Shock"
THEN add-repetitively("Check Blood Pressure", (5, min), Cnew) VALID-TIME [now, (4, hour)]

(1)

WHEN
INSERT ON clinical-findings
WITH      new.type = "Anaphylactic Shock"
THEN add ("Administer Adrenalin", Cnew) VALID-TIME [now, (10, min)]

```

(2)  Ordering heuristics,
e.g., Parallel("manual diagnostic activities", "drug administrations")

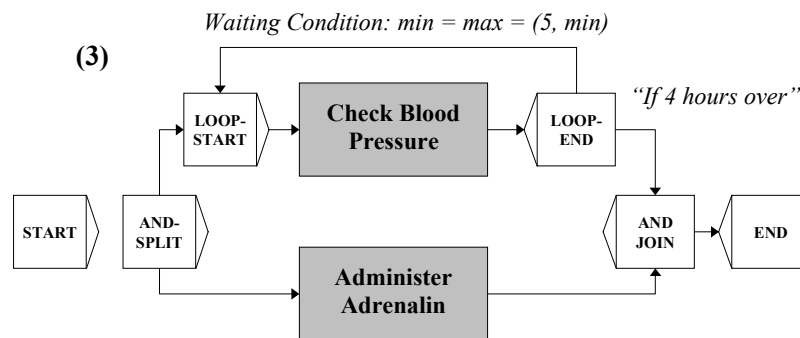


Figure 12-1: Dynamic construction of workflows.

A formal notation of the activities, ordering heuristics, and conditions has been omitted.

can only be determined at execution time when all patient data is available.

The construction of workflows could be done in a rule-based manner as follows: For every exceptional situation rules exist that describe how to behave in this situation (step (1) in Figure 12-1). When an exceptional situation occurs, the respective rules are triggered which results in a set of activities that should be executed. By using ordering heuristics such as that manual diagnostic activities (e.g., checking blood pressure) can be executed in parallel to drug administrations (2), this set of activities is then assembled to a control flow (3). By mechanisms described in Chapter 8, data flow elements then can be added to the workflow.

By this, the flexibility of rule-based systems can be combined with the strong operational support of workflow systems.

4. *Deadline Management*

So far, AGENTWORK does not address “intra-workflow” deadline management in the sense that first points in time can be assigned as deadlines to any activity node, and that second it is checked at workflow definition or execution time whether these deadlines can be met. In AGENTWORK, such deadlines can only be assigned to COMM-OUT/COMM-IN nodes to specify *inter-workflow* constraints for workflow collaboration scenarios (as described in Chapter 10). Intra-workflow deadline management has been omitted in this thesis, as this topic has already been investigated by other authors (e.g., [DADAM ET AL. 2000, BLAZEWICZ ET AL. 2001, EDER ET AL. 1999 A]).

However, for the practical usage of AGENTWORK such an intra-workflow deadline management is required as this functionality is needed for many application classes. Therefore, we plan to add it to AGENTWORK. As the estimation algorithms introduced in this thesis can be directly used to check whether the execution of a workflow part can still be finished within a given time frame, the effort to add such an intra-workflow deadline management can be viewed as manageable. Furthermore, we plan to investigate methods to automatically adapt a workflow instance if deadlines cannot be met anymore. For example, at workflow definition time some activity nodes could be marked as optional and could be automatically dropped in case deadlines may be violated.

5. *XML-Integration*

Due to the increasing importance of XML as a data interchange format, we plan to enable the AGENTWORK communication and integration layer to communicate its data in XML format, especially for inter-workflow communication. For example, we will investigate to couple CORBA with XML [VERMEULEN ET AL. 2000], or to replace CORBA by XML-based communication infrastructures such as BIZTALK [KOBIELUS 2000].

6. *Enhanced Inter-Workflow Cooperation*

As the AGENTWORK inter-workflow model obviously is simple, future work also has to concentrate on more elaborated inter-workflow cooperation. For example, based on [HEINLEIN 2001], dynamic dependencies resulting from general dependency constraints have to be considered as AGENTWORK at the moment only allows to specify static dependencies at workflow definition time.

7. *Empirical Evaluation*

The simulation environment described in Chapter 11 can be only a first step of evaluation. Therefore, we plan to integrate the AGENTWORK prototype into the HEMATOWORK project and to evaluate it empirically under real-world conditions. In particular, this means that duration values and workflow definitions have to be defined by physicians themselves (with the support of computer scientists), and that dynamic adaptations of workflow instances have to be reviewed by experienced physicians. Furthermore, it has to be checked whether measurements of activity and edge executions improve the temporal estimations of the system.

References

[A]

- AALST 1997 Aalst, W.M.P. van der (1997): *Verification of Workflow Nets*. Proceedings 18th International Conference on Application and Theory of Petri Nets (ICATPN'97), Toulouse, France (LNCS 1248). Springer, Berlin, Germany: 407-426.
- AALST 1998 Aalst, W.M.P. van der (1998): *The Application of Petri Nets to Workflow Management*. Journal of Circuits, Systems and Computers 8(1): 21-66.
- AALST 1999 Aalst, W.M.P. van der (1999): *Generic Workflow Models: How to Handle Dynamic Change and Capture Management Information?* Proceedings 4th International Conference on Cooperative Information Systems (CoopIS'99), Edinburgh, UK: 115-126.
- AALST ET AL. 2000 Aalst, W.M.P. van der; Hofstede A.H.M. ter; Kiepuszewski, B.; Barros, A.P. (2000): *Advanced Workflow Patterns*. Proceedings 5th International Conference on Cooperative Information Systems (CoopIS'2000), Eilat, Israel (LNCS 1901). Springer, Berlin, Germany: 18-29.
- ABBOT & SARIN 1994 Abbott, K.; Sarin, S.K. (1994): *Experiences with Workflow Management: Issues for the Next Generation*. Proceedings ACM Conference on Computer Supported Cooperative Work 1994, Chapel Hill, NC, USA: 113-120.
- ABDULLA & NYLÉN 2001 Abdulla, P.A.; Nylén, A. (2001): *Timed Petri Nets and BQOs*. Proceedings 22nd International Conference on Applications and Theory of Petri Nets 2001 (ICATPN'01), Newcastle upon Tyne, UK (LNCS 2075). Springer, Berlin, Germany: 53-70.
- ABROMOVITZ & ABROMOVITZ 1997 Abromovitz, H.; Abromovitz, L. (1997): *Insuring Quality: How to Improve Quality, Compliance, Customer Service, and Ethics in the Insurance Industry*. St. Lucie Press, Boca Raton, FL, USA.
- ADAM ET AL. 1998 Adam, N.R.; Atluri, V.; Huang, W.-K. (1998): *Modeling and Analysis of Workflows Using Petri Nets*. Journal of Intelligent Information Systems 10(2): 131-158.
- ADAMS & DWORKIN 1996 Adams, T.; Dworkin, S. (1996): *Workflow Interoperability between Businesses*. In: [FISCHER 2002]: 211-222.
- ALONSO ET AL. 1994 Alonso, G.; Kamath, M.; Agrawal, D.; El Abbadi, A.; Günthör, R.; C. Mohan (1994): *Failure Handling in Large Scale Workflow Management Systems*. IBM Research Report RJ 9913, San Jose, CA, USA.
- ALLEN 1984 Allen, J.F (1984): *Towards a General Theory of Action and Time*. Artificial Intelligence 23(2): 123-154.

-
- ALONSO ET AL. 1996 Alonso, G.; Agrawal, D.; El Abbadi, A.; Kamath, M.; Günthör, R.; Mohan, M. (1996): *Advanced Transaction Models in Workflow Contexts*. Proceedings 12th International Conference on Data Engineering (ICDE'1996), New Orleans, LA, USA: 574-581.
- ALONSO ET AL. 1999 Alonso, G.; Fiedler, U.; Hagen, C.; Lazcano, A.; Schuldt, H.; Weiler, N. (1999): *WISE: Business to Business E-Commerce*. In: [RIDE 1999]: 132-139.
- ALONSO & MOHAN 1997 Alonso, G.; Mohan, C. (1997): *WFMS: The Next Generation of Distributed Processing Tools*. In: [Jajodia & Kerschberg 1997]: 35-62.
- APT & OLDEROG 1994 Apt, K.R.; Olderog, E.-R. (1994): *Programmverifikation – Sequentielle, parallele und verteilte Programme*. Springer, Berlin, Germany.
- ATKINS ET AL. 1999 Atkins, E.M.; Abdelzaher, T.F.; Shin, K.G.; Durfee, E.H. (1999): *Planning and Resource Allocation for Hard Real-Time, Fault-tolerant Plan Execution*. In: [BRADSHAW ET AL. 1999]: 244-251.
- ATTIE ET AL. 1996 Attie, P.; Singh, M.P.; Emerson, E.A.; Sheth, A.; Rusinkiewicz, M. (1996): *Scheduling Workflows by Enforcing Intertask Dependencies*. Distributed Systems Engineering Journal 3(4): 222-238.
- AYLETT ET AL. 2000 Aylett, R.S.; Petley, G.J.; Chung, P.W.H.; Chen, B.; Edwards, D. W. (2000): *AI Planning: Solutions for Real-World Problems*. Knowledge-Based Systems 13 (2-3): 61-69.

[B]

- BAETEN & WEIJLAND 1990 Baeten, J.C.M.; Weijland, W.P. (1990): *Process Algebra*. Cambridge University Press, Cambridge, UK.
- BAKER 1997 Baker, S. (1997): *CORBA Distributed Objects*. Addison Wesley, Reading, MA, USA.
- BAKER ET AL. 1999 Baker, D.; Georgakopoulos, D.; Schuster, H.; Cassandra, A.R.; Cichocki, A. (1999): *Providing Customized Process and Situation Awareness in the Collaboration Management Infrastructure*. Proceedings 4th International Conference on Cooperative Information Systems (CoopIS'99), Edinburgh, UK: 79-91.
- BARALIS 1999 Baralis, E.: *Rule Analysis*. In: [PATON 1999]: 51-67.
- BARBARÁ ET AL. 1996 Barbará, D.; Mehrotra, S.; Rusinkiewicz, M. (1996): *INCAS: Managing Dynamic Workflows in Distributed Environments*. Journal of Database Management 7(1): 5-15.
- BAUDINET ET AL. 1993 Baudinet, M.; Chomicki, J.; Wolper, P. (1993): *Temporal Deductive Databases*. In: [TANSEL ET AL. 1993]: 294-320.
- BAUER & DADAM 1997 Bauer, Th.; Dadam, P. (1997): *A Distributed Execution Environment for Large-Scale Workflow Management Systems with Subnets and Server Migration*. Proceedings 2nd International Conference on Cooperative Information Systems (CoopIS'97), Kiawah Island, SC, USA: 99-108.
- BAUER & DADAM 1999 Bauer, Th.; Dadam, P. (1999): *Verteilungsmodelle für Workflow-Management-Systeme - Klassifikation und Simulation*. Informatik Forschung und Entwicklung 14 (4): 203-217.
- BAUER & DADAM 2000 Bauer, Th.; Dadam, P. (2000): *Efficient Distributed Workflow Management with Variable Server Assignments*. Proceedings 12th International Conference on Advanced Information Systems Engineering (CAiSE'2000), Stockholm, Sweden (LNCS 1789). Springer, Berlin, Germany: 94-109.

-
-
- BAUER ET AL. 2001 Bauer, Th.; Reichert, M.; Dadam, P. (2001): *Adaptives und verteiltes Workflow-Management*. Proceedings 9th Bi-Annual German Database Conference (BTW'2001), Oldenburg, Germany: 47-66.
- BAUMGARTEN 1996 Baumgarten, B. (1996): *Petri-Netze: Grundlagen und Anwendungen*. Spektrum Akademischer Verlag, Heidelberg, Germany.
- BECKSTEIN & KLAUSNER 1999A Beckstein, C.; Klausner, J. (1999): *A Planning Framework for Workflow Management*. Proceedings International Workshop on Intelligent Workflow and Process Management (at IJCAI-99), Stockholm, Sweden.
- BECKSTEIN & KLAUSNER 1999B Beckstein, C.; Klausner, J. (1999): *A Meta Level Architecture for Workflow Management*. Transactions of the Society for Design and Process Science (Journal of Integrated Design and Process Science) 3(1): 15-26.
- BELAZZI ET AL. 1999 Bellazzi, R.; Larizza, Ch.; Magni, P.; Montani, St.; De Nicolao, G. (1999): *Intelligent Analysis of Clinical Time Series by Combining Structural Filtering and Temporal Abstractions*. Proceedings Joint European Conference on Artificial Intelligence in Medicine and Medical Decision (AIMMD'99), Aalborg, Denmark (LNCS 1620). Springer, Berlin, Germany: 261-270.
- BEMMEL & MUSEN 1997 Bommel, van J.H.; Musen, M. (Eds.) (1997): *Handbook of Medical Informatics*. Springer, Berlin, Germany.
- BENTHEM 1995 Benthem; J. van (1995): *Temporal Logic*. In: [GABBAY ET AL. 1995]: 242-350.
- BERGER 1997 Berger, H. (1997): *TherPlan: Ein Therapieplan-Editor für die Hämato-Onkologie (TherPlan: A Therapy Editor for Hematooncology)*. Diploma thesis, University of Leipzig, Germany.
- BERGER & TUZHILIN 1998 Berger, G.; Tuzhilin, A. (1998): *Discovering Unexpected Patterns in Temporal Data Using Temporal Logic*. In: [ETZION ET AL. 1998]: 281-309.
- BERNDT & CLIFFORD 1996 Berndt, D.; Clifford, J. (1996): *Finding Patterns in Time Series: A Dynamic Programming Approach*. In: Advances in Knowledge Discovery and Data Mining 1996. AAAI/MIT Press, Cambridge, MA, USA: 229-248.
- BERNSTEIN ET AL. 1990 Bernstein, P.A.; Hsu, M.; Mann, B (1990): *Implementing Recoverable Requests Using Queues*. Proceedings ACM International Conference on Management of Data (SIGMOD'90), Atlantic City, NJ, USA: 112-122.
- BERTHOLD ET AL. 1999 Berthold, A.; Mende, U.; Schuster, H. (1999): *SAP Business Workflow. Konzept, Anwendung, Entwicklung*. Addison Wesley, Bonn, Germany.
- BLAKELEY ET AL. 1995 Blakeley, J.A.; Harris, H.; Lewis, R. (1995): *Messaging and Queuing Using the MQI*. McGraw-Hill, Columbus, OH, USA.
- BLAZEWICZ ET AL. 2001 Blazewicz, J.; Ecker, K.-H.; Pesch, P.; Schmidt, G.; Weglarz, J. (2001): *Scheduling Computer and Manufacturing Processes*. Springer, Berlin, Germany.
- BLUM & FURST 1997 Blum, A.; Furst, M.L. (1997): *Fast Planning Through Planning Graph Analysis*. Artificial Intelligence 90(1-2): 281-300.
- BÖHLEN & JENSEN 1996 Böhlen, M.; Jensen, C. S. (1996): *Seamless Integration of Time into SQL*. Technical Report R-96-2049, Aalborg University, Department of Computer Science, Denmark.

-
-
- BÖHME 2000 Böhme, R. (2000): *Konzeption und Implementierung eines Workflow-Editors (Concept and Implementation of a Workflow Editor)*. Diploma thesis, University of Leipzig, Germany.
- BONNER & KIFER 1994 Bonner, A.J.; Kifer, M. (1994): *An Overview of Transaction Logic*. Theoretical Computer Science 133(2): 205-265.
- BONNER & KIFER 1998 Bonner, A.J.; Kifer, M. (1998): *The State of Change: A Survey*. In: Freitag, B.; Decker, H.; Kifer, M.; Voronkov, A. (Eds.): *Transactions and Change in Logic Databases* (LNCS 1472). Springer, Berlin, Germany: 1-36.
- BORGIDA & MURATA 1999 Borgida, A.; Murata, T. (1999): *Tolerating Exceptions in Workflows: a Unified Framework for Data and Processes*. In: [GEORGAKOPOULOS ET AL. 1999 A]: 59-68.
- BRACHMAN 1977 Brachman, R.J. (1977): *What's in a Concept: Structural Foundations for Semantic Networks*. International Journal of Man-Machine Studies 9: 127-152.
- BRADSHAW ET AL. 1999 Bradshaw, J.; Etzioni, O.; Müller, J. (Eds.) (1999): *AGENTS '99*. Proceedings 3rd International Conference on Autonomous Agents, Seattle, WA, USA.
- BREITBART ET AL. 1993 Breitbart, Y.; Deacon, A.; Schek, H.-J.; Sheth, A.; Weikum, G. (1993): *Merging Application-Centric and Data-Centric Approaches to Support Transaction-Oriented Multi-System Workflows*. SIGMOD Record 22(3): 23-30.
- BREWKA 1996 G. Brewka (Ed.) (1996): *Principles of Knowledge Representation*. CSLI Publications, Studies in Logic, Language and Information, Stanford, CA, USA.
- BRICON-SOUF ET AL. 1998 Bricon-Souf, N.; Renard, J.M.; Beuscart, R. (1998): *Dynamic Workflow for Complex Activity in Intensive Care Unit*. Proceedings 9th World Congress on Medical Informatics (MED-INFO'98), Seoul, South-Korea: 227-231.
- BROSTEANU ET AL. 1998 Brostenanu, O.; Klöss, M.; Speer, R.; Hasenclever, D. (1998): *Standardisierte Vorgehensweise für klinische Studien am Institut für Medizinische Informatik, Statistik und Epidemiologie*. University of Leipzig, Germany.
- BROVERMAN & CROFT 1987 Broverman, C.A.; Croft, B.A. (1987): *Reasoning about Exceptions during Plan Execution Monitoring*. Proceedings 6th National Conference on Artificial Intelligence (AAAI'87). Seattle, WA, USA: 190-195.
- BRUCKER ET AL. 1999 Brucker, P.; Hilbig, T.; Hurink, J. (1999): *A Branch & Bound Algorithm for Scheduling Problems with Positive and Negative Time Lags*. Discrete Applied Mathematics 94: 77-99.
- BRÜMMER 1997 Brümmer, F. (1997): *Entwurf und Implementation einer Patienten-Datenbank zur Repräsentierung onkologischer Behandlungsverläufe (Concept and Implementation of a Patient Database for the Representation of Cancer Treatments)*. Diploma thesis, University of Leipzig, Germany.
- BUCHANAN & SHORTLIFFE 1984 Buchanan, B.G.; Shortliffe, E.H. (1984): *Rule-Based Expert Systems: The MYCIN Experiments of the Stanford Heuristic Programming Project*. Addison Wesley, Reading, MA, USA.
- BUKHRES ET AL. 1998 Bukhres, O.; Eder, J.; Salza, S. (Eds.) (1998): *Proceedings International EDBT98 Workshop on Workflow Management Systems (EDBT-WFMS'98)*. Valencia, Spain.
- BUSSLER 1998 Bussler, Ch.: *Workflow Interoperability Classification and its Implication to Workflow Management System Architectures*. In: [BUKHRES ET AL. 1998]: 45-54.

[C]

- CALVANESE ET AL. 1998 Calvanese, D.; Lenzerini, M.; Nardi, D. (1998): *Description Logics for Conceptual Data Modeling*. In: [CHOMICKI & SAAKE 1998]: 229-263.
- CAOQUETTE 1998 Caouette, J.B.; Altman, E.I.; Narayanan, P. (1998): *Managing Credit Risk: The Next Great Financial Challenge*. Wiley, New York, NY, USA.
- CARSI ET AL 1998 Carsi J.A.; Letelier P.; Sanchez P. (1998): *A DOOD System for Treating the Schema Evolution Problem*. Proceedings Demo Session at EDBT'98. Valencia, Spain.
- CASATI ET AL. 1996 Casati, F.; Ceri, S.; Pernici, B.; Pozzi, G. (1996): *Semantic Workflow Interoperability*. Proceedings 5th International Conference on Extending Database Technology (EDBT'96), Avignon, France (LNCS 1057). Springer, Berlin, Germany: 443-462.
- CASATI 1998 Casati, F. (1998): *Models, Semantics and Formal Methods for the Design of Workflows and their Exceptions*. PhD thesis, Politecnico di Milano, Italy.
- CASATI ET AL. 1998 Casati, F.; Ceri, S.; Pernici, B.; Pozzi, G. (1998): *Workflow Evolution*. Data & Knowledge Engineering 24(3): 211-238.
- CASATI 1999 Casati, F. (1999): *Semantic Interoperability in Interorganizational Workflows*. In: [LUDWIG ET AL. 1999].
- CASATI ET AL. 1999 Casati, F.; Ceri, S.; Paraboschi, S.; Pozzi, G. (1999): *Specification and Implementation of Exceptions in Workflow Management Systems*. ACM Transactions on Database Systems 24(3): 405-451.
- CASATI & POZZI 1999 Casati, F.; Pozzi, G. (1999): *Modeling Exceptional Behaviors in Commercial Workflow Management Systems*. Proceedings 4th International Conference on Cooperative Information Systems (CoopIS'99), Edinburgh, UK: 127-138.
- CATTELL ET AL. 2000 Cattell, R. G. G.; Barry, D. K.; Berler, M.; Eastman, J.; Jordan, D.; Russell, C.; Schadow, O.; Stanienda, T.; Velez, F. (2000): *The Object Data Standard: ODMG 3.0*. Morgan Kaufmann, San Francisco, CA, USA
- CERI ET AL. 1989 Ceri, S.; Gottlob, G.; Tanca, L. (1989): *What you Always Wanted to Know About Datalog (And Never Dared to Ask)*. Transactions on Data and Knowledge Engineering 1(1): 146-166.
- CERRITO & MAYER 1998 Cerrito, S.; Mayer, M.C. (1998): *Using Linear Temporal Logic to Model and Solve Planning Problems*. Proceedings 8th International Conference of Artificial Intelligence: Methodology, Systems, and Applications (AIMSA'98), Sozopol, Bulgaria (LNCS 1480). Springer, Berlin, Germany: 141-152.
- CHAKRAVARTHY ET AL. 1994 Chakravarthy, S.; Krishnaprasad, V.; Anwar, E.; Kim, S-K. (1994): *Composite Events for Active Databases: Semantics Contexts and Detection*. Proceedings 20th International Conference on Very Large Databases (VLDB'94), Santiago, Chile: 606-617.
- CHAPMAN 1987 Chapman, D. (1987): *Planning for Conjunctive Goals*. Artificial Intelligence 32(3): 333-377.
- CHEN ET AL. 1993 Chen, W.; Kifer, M.; Warren, D.S. (1993): *HiLog: A Foundation for Higher-Order Logic Programming*. Journal of Logic Programming 15(3): 187-230.
- CHIU ET AL. 1999 Chiu, D.K.W.; Li, Q.; Karlapalem, K. (1999): *A Meta Modeling Approach to Workflow Management System Supporting Exception Handling*. Information Systems 24(2): 159-184.

-
- CHOMICKI 1994 Chomicki, J. (1994): *Temporal Query Languages: A Survey*. Proceedings International Conference on Temporal Logic (ICTL'94), Bonn, Germany: 506-534.
- CHOMICKI & SAAKE 1998 Chomicki, J.; Saake, G. (Eds.) (1998): *Logics for Databases and Information Systems*. Kluwer, New York, NY, USA.
- CHOMICKI & TOMAN 1998 Chomicki, J.; Toman, D. (1998): *Temporal Logic in Information Systems*. In: [CHOMICKI & SAAKE 1998]: 31-70.
- CLANCEY 1985 Clancey, W.J. (1985): *Heuristic Classification*. Artificial Intelligence 27(3): 289-350.
- CLIFFORD ET AL. 1997 Clifford, J.; Dyreson, C.; Isakowitz, T.; Jensen, C.S.; Snodgrass, R.T. (1997): *On the Semantics of "Now" in Databases*. ACM Transactions on Database Systems 22(2): 171-214.
- CLYMER 1993 Clymer, J.R. (1993): *System Design and Evaluation Using Discrete Event Simulation with Artificial Intelligence*. Proceedings ACM Conference on Winter Simulation, Los Angeles, CA, USA: 1347-1356.
- CODD 1970 Codd, E.F. (1970): *A Relational Model of Data for Large Shared Data Banks*. Communications of the ACM 13(6): 377-387.
- COLLET ET AL. 1998 Collet, C.; Vargas-Solar, G.; Grazziotin-Ribeiro, H. (1998): *Towards a Semantic Event Service for Distributed Active Database Applications*. Proceedings 9th International Conference on Database and Expert Systems Applications, Vienna, Austria (LNCS 1460). Springer, Berlin, Germany: 16-27.
- COMBI & CHITTARO 1999 Combi, C.; Chittaro, L. (1999): *Abstraction on Clinical Data Sequences: An Object-Oriented Data Model and a Query Language Based on the Event Calculus*. Artificial Intelligence in Medicine 17(3): 271-301.

[D]

- DADAM ET AL. 2000 Dadam, P.; Reichert, M.; Kuhn, K. (2000): *Clinical Workflows - The Killer Application for Process-oriented Information Systems?* Proceedings 4th International Conference on Business Information Systems (BIS'2000), Poznan, Poland. Springer, Berlin, Germany: 36-59.
- DADAM & KLAS 1997 Dadam, P.; Klas, W. (1997): *The Database and Information System Research Group at the University of Ulm*. SIGMOD Record 26(4): 75-79.
- DADUNA 2001 Daduna, H. (2001): *Queueing Networks with Discrete Time Scale - Explicit Expressions for the Steady State Behavior of Discrete Time Stochastic Networks*. Springer, Berlin, Germany.
- DATE & DARWEN 1997 Date, C.J.; Darwen H. (1997): *A Guide to SQL*. Addison Wesley, Reading, MA, USA.
- DAVIS ET AL. 1996 Davis, J.; Du, W.; Shan, M.-C.; Dayal, U. (1996): *Flexible Compensation of Workflow Processes*. Technical Report HPL-96-72. Hewlett-Packard Software Technology Laboratory. (<http://www.hpl.hp.com/techreports>)
- DAVULCU ET AL. 1999 Davulcu, H.; Kifer, M.; Pokorny, L.R.; Ramakrishnan, C.R.; Ramakrishnan, I.V.; Dawson, S. (1999): *Modeling and Analysis of Interactions in Virtual Enterprises*. In: [RIDE 1999]: 12-18.
- DAVULCU ET AL. 1998 Davulcu, H.; Kifer, M.; Ramakrishnan, C.R.; Ramakrishnan, I.V. (1998): *Logic-Based Modeling and Analysis of Workflows*. Proceedings 17th ACM Symposium on Principles of Database Systems (PODS'98), Seattle, WA, USA: 25-33.

-
-
- DAYAL ET AL. 1991 Dayal, U.; Hsu, M.; Ladin, R. (1991): *A Transactional Model for Long-running Activities*. Proceedings of 17th International Conference on Very Large Data Bases (VLDB'91), Barcelona, Spain: 113-122.
- DAYAL ET AL. 1996 Dayal, U.; Buchmann, A.P.; Chakravarthy, S. (1996): The HiPAC Project. In: [WIDOM & CERI 1996]: 177-206.
- DECHTER ET AL. 1991 Dechter, R.; Meiri, I.; Pearl, J. (1991): *Temporal Constraint Networks*. Artificial Intelligence 49 (1-3): 61-95.
- DEJONG & BENNETT 1997 DeJong, G.F.; Bennett, S.W. (1997): *Permissive Planning: Extending Classical Planning to Uncertain Task Domains*. Artificial Intelligence 89 (1-2): 173-217.
- DELGRANDE ET AL. 1999 Delgrande, J.P.; Gupta, A.; Van Allen, T. (1999): *Point-Based Approaches to Qualitative Temporal Reasoning*. Proceedings of 16th National Conference on Artificial Intelligence and Eleventh Conference on Innovative Applications of Artificial Intelligence (AAAI/IAAI'99), Orlando, FL, USA: 739-744.
- DELLAROCAS & KLEIN 2000 Dellarocas C.; Klein M. (2000): *An Experimental Evaluation of Domain-Independent Fault Handling Services in Open Multi-Agent Systems*. Proceedings of International Joint Conference on Multi-Agent Systems (ICMAS-2000), Boston, MA, USA.
- DELLEN ET AL. 1997 Dellen, B.; Pews, G.; Maurer, F. (1997): *Knowledge Based Techniques to Increase the Flexibility of Workflow Management*. Data & Knowledge Engineering Journal 23(3): 269-295.
- DEUX 1991 Deux, O (1991): *The O2 System*. Communications of the ACM 34(10): 34-48.
- DEVITA ET AL. 1987 De Vita, V.T.; Hubbard, S.M.; Longo, D.L. (1987): *The Chemotherapy of Lymphomas: Looking Back, Moving Forward - The Richard and Hinda Rosenthal Foundation Award Lecture*. Cancer Research 47: 5810-5824.
- DIEHL 1993 Diehl, V. (Ed.) (1993): *Deutsche Hodgkin Lymphom Studiengruppe - Studienprotokolle der Primärtherapie HD4, HD8, HD9 (German Hodgkin Disease Study Group - Trial Protocols of Primary Therapy HD4, HD8, HD9)*. University of Cologne, Germany.
- DIEHL ET AL. 1998 Diehl, V.; Franklin, J.; Hasenclever, D.; Tesch, H.; Pfreundschuh, M.; Lathan, B.; Paulus, U.; Sieber, M.; Rueffer, J.U.; Sextro, M.; Engert, A.; Wolf, J.; Hermann, R.; Holmer, L.; Stappert-Jahn, U.; Winnerlein-Trump, E.; Wulf, G.; Krause, S.; Glunz, A.; von, Kalle, K.; Bischoff, H.; Haedicke, C.; Duehmke, E.; Georgii, A.; Löffler, M. (1998): *BEACOPP, a New Dose-Escalated and Accelerated Regimen, is at least as Effective as COPP/ABVD in Patients with Advanced-Stage Hodgkin's Lymphoma: Interim Report from a Trial of the German Hodgkin's Lymphoma Study Group*. Journal of Clinical Oncology 16(12): 3810-21.
- DIETZSCH 2000 Dietzsch, A. (2000): *Konzeption und Implementierung einer Workflow-Engine (Concept and Implementation of a Workflow Engine)*. Diploma thesis, University of Leipzig, Germany.
- DINN ET AL. 1999 Dinn, A.; Paton, N.W.; Williams, M.H. (1999): *Active Rule Analysis and Optimisation in the ROCK & ROLL Deductive Object-Oriented Database*. Information Systems 24(4), 327-353.
- DIJKSTRA 1968 Dijkstra, E. W. (1968): *Co-Operating Sequential Processes*. In: Genuys, F. (Ed.): *Programming Languages*. Academic Press, San Diego, CA, USA: 43-112.
- DOGAC ET AL. 1998 Dogac, A.; Kalinichenko, L.; Ozsu, T.; Sheth, A. (Eds.) (1998): *Workflow Management Systems and Interoperability*. Springer, Berlin, Germany.

DRABBLE & TATE 1995 Drabble, B.; Tate, A. (1995): *O-Plan: A Situated Planning Agent*. Proceedings 3rd European Workshop on Planning Systems (EWSP-95), Assisi, Italy.

[E]

EDDON 1999 Eddon, G.; (1999): *COM+: The Evolution of Component Services*. IEEE Computer 32(7): 104-106.

EDER ET AL. 1999 A Eder, J.; Panagos, T.; Rabinovich, R. (1999): *Time Constraints in Workflow Systems*. Proceedings 11th International Conference on Advanced Information Systems Engineering (CAiSE'99), Heidelberg, Germany (LNCS 1626). Springer, Berlin, Germany: 286-300.

EDER ET AL. 1999 B Eder, J.; Panagos, T.; Pozewaunig, H.; Rabinovich, R. (1999): *Time Management in Workflow Systems*. Proceedings 3rd International Conference on Business Information Systems (BIS'99), Poznan, Poland. Springer, Berlin, Germany: 265-280.

EDER & LIEHART 1996 Eder, J.; Liebhart, W. (1996): *Workflow Recovery*. Proceedings First International Conference on Cooperative Information Systems (CoopIS'96), Brussels, Belgium: 124-134.

EDMOND & HOFSTEDE 2000 Edmond, D.; Hofstede, A.H.M. ter (2000): *A Reflective Infrastructure for Workflow Adaptability*. Data & Knowledge Engineering 34(3): 271-304.

EISELT & FRAJER 1977 Eiselt, H.A.; Frajer, H.v. (1977): *Operations Research Handbook: Standard Algorithms and Methods with Examples*. De Gruyter, Berlin, Germany.

EISENBERG & MELTON 1999 Eisenberg, A.; Melton, J. (1999): *SQL: 1999, formerly known as SQL 3*. SIGMOD Record 28(1): 131-138.

EISENBERG & MELTON 2000 Eisenberg, A.; Melton, J. (2000): *SQL Standardization: The Next Steps*. SIGMOD Record 29(1): 63-67.

ELLIS ET AL. 1995 Ellis, C.; Keddara, K.; Rozenberg, G. (1995): *Dynamic Change Within Workflow Systems*. Proceedings ACM Conference on Organizational Computing Systems (COOCS'95), Milpitas, CA, USA: 10-21.

ELLIS ET AL. 1998 Ellis, C.A.; Keddara, K.; Wainer, W. (1998): *Modeling Workflow Dynamic Changes Using Timed Hybrid Flow Nets*. Proceedings Workshop on Workflow-Management at 19th International Conference on Applications and Theory of Petri Nets (ICATPN'98), Lisbon, Portugal: 109-128.

ELLIS & MALTZAHN 1997 Ellis, C.A.; Maltzahn, C. (1997): *The Chautauqua Workflow System*. Proceedings 30th Hawaii International Conference on System Sciences (HICSS'97), Maui, HI, USA.

ELMARGARMID 1992 Elmagarmid, A.K. (Ed.) (1992): *Database Transaction Models for Advanced Applications*. Morgan Kaufmann, San Fransisco, CA, USA.

ELMARGARMID & DU 1998 Elmagarmid, A.K.; Du, W. (1998): *Workflow Management: State of the Art vs. State of the Products*. In: [DOGAC ET AL. 1998]: 1-17.

ETZION ET AL. 1998 Etzion, O.; Jajodia, S.; Sripada, S.M. (Eds.) (1998): *Temporal Databases: Research and Practice* (LNCS 1399). Springer, Berlin, Germany.

EVANS & EDWARD 1992 Evans, J.; Edward, M. (1992): *Optimization Algorithms for Networks and Graphs*. M. Dekker, New York, NY, USA.

[F]

- FEILER 2000 Feiler, J. (2000): *Application Servers*. Morgan Kaufmann, San Fransisco, CA, USA.
- FERNANDES 1999 Fernandes, A.A. (1999): *Comparing Deductive and Active Databases*. In: [PATON 1999]: 177-194.
- FIEBIG 1999 Fiebig, F. (1999): *Geschäftsprozeß-Modellierung und -Simulation in einer onkologischen Studienzentrale (Business Process Modelling and Simulation at an Oncological Commission)*. Diploma thesis, University of Leipzig, Germany.
- FIKES & NILSSON 1971 Fikes, R. E.; Nilsson, N. J. (1971): *STRIPS: A New Approach to the Application of Theorem Proving to Problem Solving*. Artificial Intelligence 2, 189-208.
- FISCHER 2002 Fischer, L. (2002): *The WfMC Workflow Handbook 2002*. Future Strategies, FL, USA.
- FOWLER & SCOTT 1998 Fowler, M.; Scott, K. (1998): *UML distilled: Applying the Standard Object Modeling Language*. Addison Wesley, Reading, MA, USA.
- FRANCONI 2002 Franconi, E. (2002): *Description Logics for Natural Language Processing*. In: Baader, F.; McGuinness, D.L.; Nardi, D.; Patel-Schneider, P.F (Eds.): *Description Logics Handbook*. Cambridge University Press, Cambridge, UK.
- FRANKLIN & GRAESSER 1997 Franklin, S.P.; Graesser, A. (1997): *Is it an Agent, or just a Program? A Taxonomy for Autonomous Agents*. In: [MÜLLER ET AL. 1997 C]: 21-35.
- FRIDSMA ET AL. 1996 Fridsma, D.B.; Gennari, J.H.; Musen, M.A. (1996): *Making Generic Guidelines Site-Specific*. Proceedings American Medical Informatics Association Annual Fall Symposion 1996, Washington, D.C., USA: 597-601.
- FROHN ET AL. 1997 Frohn, J.; Himmeröder, R.; Kandzia, P.-Th.; Lausen, G.; Schlepphorst, C. (1997): *FLORID - Ein Prototyp für F-Logik (A prototype for F-Logic)*. Proceedings 7th Bi-Annual German Database Conference (BTW'97), Ulm, Germany: 100-117.

[G]

- GABBAY 1987 Gabbay, D.M. (1987): *Modal and Temporal Logic Programming*. In: Galton, A. (Ed.): *Temporal Logics and their Applications*. Academic Press, San Diego, CA, USA: 197-236.
- GABBAY ET AL. 1995 Gabbay, D.M.; Hogger, C. J.; Robinson, J. A. (1995): *Handbook of Logic in Artificial Intelligence and Logic Programming, Volume 4: Epistemic and Temporal Reasoning*. Oxford University Press, Oxford, UK.
- GABBAY ET AL. 1998 Gabbay, D.M. et al. (1998): *Handbook of Logic in Artificial Intelligence and Logic Programming, Volume 1-5*. Oxford University Press, Oxford, UK: 1992-1998.
- GABBAY & MCBRIEN 1991 Gabbay, D.M.; McBrien, P. (1991): *Temporal Logic and Historical Databases*. Proceedings 17th International Conference on Very Large Data Bases (VLDB'91), Barcelona, Spain: 423-430.
- GADIA 1988 Gadia, S. (1988): *A Homogeneous Relational Model and Query Languages for Temporal Databases*. ACM Transactions on Database Systems 13(4): 418-448.
- GARCIA-MOLINA & SALEM 1987 Garcia-Molina, H.; Salem, S. (1987): *Sagas*. Proceedings ACM International Conference on Management of Data (SIGMOD'87), San Francisco, CA, USA: 249-259.

-
-
- | | |
|---------------------------------|---|
| GARCIA-MOLINA ET AL. 1991 | Garcia-Molina, H.; Gawlick, D.; Klein, J.; Kleissner, K.; Salem, K. (1991): <i>Modeling Long-Running Activities as Nested Sagas</i> . Data Engineering Bulletin 14(1): 14-18. |
| GEHANI ET AL. 1992 | Gehani, N.; Jagadish, H. V.; Smueli, O. (1992): <i>Composite Event Specification in Active Databases: Model and Implementation</i> . Proceedings 18th International Conference on Very Large Data Bases (VLDB'92), Vancouver, Canada: 327-338. |
| GEORGAKOPOULOS ET AL. 1995 | Georgakopoulos, D.; Hornick, M.; Sheth, A. (1995): <i>An Overview of Workflow Management: From Process Modeling to Infrastructure for Automation</i> . Journal on Distributed and Parallel Database Systems 3 (2): 119-153. |
| GEORGAKOPOULOS & HORNICK 1994 | Georgakopoulos, D.; Hornick, M. (1994): <i>A Framework for Enforceable Specification of Extended Transaction Models and Transactional Workflows</i> . International Journal of Cooperative Information Systems 3(3): 599-617. |
| GEORGAKOPOULOS ET AL. 1999 A | Georgakopoulos, D.; Prinz, W.; Wolf, A. (Eds.) (1999): <i>Proceedings of International Joint Conference on Work Activities Coordination and Collaboration (WACC '99)</i> , San Francisco, CA, USA. ACM Press, New York, NY, USA. |
| GEORGAKOPOULOS ET AL. 1999 B | Georgakopoulos, D.; Schuster, H.; Cichocki, A.; Baker, D. (1999): <i>Managing Process and Service Fusion in Virtual Enterprises</i> . Information Systems 24(6): 429-456. |
| GEORGAKOPOULOS ET AL. 2000 | Georgakopoulos, D.; Schuster, H.; Baker, D.; Cichocki, A. (2000): <i>Managing Escalation of Collaboration Processes in Crisis Mitigation Situations</i> . Proceedings 16th International Conference on Data Engineering (ICDE'00), San Diego, CA, USA: 45-56. |
| GEPPERT ET AL. 1996 | Geppert, A.; Berndtsson, M.; Lieuwen, D.; Zimmermann, J. (1996): <i>Performance Evaluation of Active Database Management Systems using the BEAST Benchmark</i> . Technical Report CS 96.01, University of Zurich, Switzerland. |
| GEPPERT ET AL. 1998 | Geppert, A.; Tombros, D.; Dittrich K.R (1998): <i>Defining the Semantics of Reactive Components in Event-Driven Workflow Execution with Event Histories</i> . Information Systems 23(3): 235-252. |
| GIARRATANO & RILEY 1993 | Giarratano, J.; Riley, G. (1993): <i>Expert Systems: Principles and Programming</i> . PWS Publishing Company, Boston, MA, USA. |
| GIBLIN & LAM 2000 | Giblin, G.; Lam, R. (2000): <i>Programming Workflow Applications With Domino</i> . CMP Books, Gilroy, CA, USA. |
| GIL & MELZ 1996 | Gil, Y.; Melz, E. (1996): <i>Explicit Representations of Problem-Solving Strategies to Support Knowledge Acquisition</i> . Proceedings 13th National Conference on Artificial Intelligence and 8th Conference on Innovative Applications of Artificial Intelligence (AAAI/IAAI'96), Portland, OR, USA, (vol. 1): 469-476. |
| GÖTTLER 1988 | Göttler, H. (1988): <i>Graphgrammatiken in der Softwaretechnik (Graph-Grammars in Software Engineering)</i> . Springer, Berlin, Germany. |
| GRAY & REUTER 1993 | Gray, J.; Reuter, A. (1993): <i>Transaction Processing: Concepts and Techniques</i> . Morgan Kaufmann, San Fransisco, CA, USA. |
| GREFEN & REMMERTS DE VRIES 1998 | Grefen, P.; Remmerts de Vries, R. (1998): <i>A Reference Architecture for Workflow Management Systems</i> . Data & Knowledge Engineering 27(1): 31-57. |

-
-
- GREFEN ET AL. 1999 A Grefen, P.; Pernici, B.; Sánchez, G. (Eds.) (1999): *Database Support for Workflow Management: The WIDE Project*. Kluwer, New York, NY, USA.
- GREFEN ET AL. 1999 B Grefen, P.; Vonk, J.; Boertjes, E.; Apers, P. (1999): *Semantics and Architecture of Global Transaction Support in Workflow Environments*. Proceedings 4th International Conference on Cooperative Information Systems (CoopIS'99), Edinburgh, UK: 348-359.
- GREFEN & HOFFNER 1999 Grefen, P.; Hoffner, Y. (1999): *CrossFlow: Cross-Organizational Workflow Support for Virtual Organizations (Extended Abstract)*. In: [RIDE 1999]: 90-91.
- GREFEN ET AL. 2000 Grefen, P.; Aberer, K.; Hoffner, Y.; Ludwig, H. (2000): *CrossFlow: Cross-Organizational Workflow Management in Dynamic Virtual Enterprises*. International Journal of Computer Systems Science & Engineering 15(5): 277-290.
- GREINER 2000 Greiner, U. (2000): *Konzeption und Implementierung eines Agenten zur ereignisorientierten Adaptation von Workflows (Concept and Implementation of an Agent for Event-Oriented Workflow Adaptation)*. Diploma thesis, University of Leipzig, Germany.
- GRONEMANN ET AL. 1999 Gronemann, B.; Joeris, G.; Scheil, S.; Steinfort, M.; Wache, H. (1999): *Supporting Cross-Organizational Engineering Processes by Distributed Collaborative Workflow Management - The MOKASSIN Approach*. Proceedings 2nd Symposium on Concurrent Multidisciplinary Engineering (CME'99) / 3rd International Conference on Global Engineering Networking (GEN'99), Bremen, Germany.
- GROSS & LÖFFLER 1998 Gross, R.; Löffler, M. (1998): *Prinzipien der Medizin. Eine Übersicht ihrer Grundlagen und Methoden*. Springer, Berlin, Germany.
- [H]
- HAAKE & WANG 1997 Haake, J.M.; Wang, W. (1997): *Flexible Support for Business Processes: Extending Cooperative Hypermedia with Process Support*. Proceedings International ACM SIGGROUP Conference on Supporting Group Work 1997, Phoenix, AZ, USA: 341-350.
- HAGEMEYER ET AL. 1997 Hagemeyer, J.; Herrmann, Th.; Just-Hahn, K.; Striemer, R. (1997): *Flexibilität bei Workflow-Management-Systemen*. Proceedings of Software-Ergonomie '97, Dresden, Germany. Teubner, Stuttgart, Germany
- HAGEN & ALONSO 1998 Hagen, C.; Alonso, G. (1998): *Flexible Exception Handling in the OPERA Process Support System*. Proceedings 18th International Conference on Distributed Computing Systems (ICDCS'98), Amsterdam, The Netherlands: 526-533.
- HAGEN & ALONSO 1999 Hagen, C.; Alonso, G. (1999): *Beyond the Black Box: Event-based Inter-Process Communication in Process Support Systems*. Proceedings 19th International Conference on Distributed (ICDCS'99). Austin, TX, USA: 450-457.
- HALL & SHAHMEHRI 1996 Hall, T.; Shahmehri, N. (1996): *An Intelligent Multi-Agent Architecture for Support of Process Reuse in a Workflow Management System*. Proceedings 1st International Conference on the Practical Application of Intelligent Agents and Multi-Agent Technology (PAAM '96), London, UK: 331-343.
- HAMMER & STANTON 1995 Hammer, M.; Stanton, S.A. (1995): *The Reengineering Revolution – The Handbook*. Harper Collins, New York, NY, USA.
- HAMMOND 1990 Hammond, K. J. (1990): *Explaining and Repairing Plans That Fail*. Artificial Intelligence 45: 173-228.

-
-
- HAN ET AL. 1996 Han, Y.; Himmighöfer, J.; Schaaf, T.; Wikarski, D. (1996): *Management of Workflow Resources to Support Runtime Adaptability and System Evolution*. Proceedings First International Conference on Practical Aspects of Knowledge Management (PAKM'96), Basel, Switzerland.
- HANSON ET AL. 1998 Hanson, E.N.; Chen, I.-C.; Dastur, R.; Engel, K.; Ramaswamy, V.; Tan, W.; Xu, C. (1998): *A Flexible and Recoverable Client/Server Database Event Notification System*. VLDB Journal 7(1): 12-24.
- HÄRDER & RAHM 2001 Härder, T.; Rahm, E. (2001): *Datenbanksysteme - Konzepte und Techniken der Implementierung (2nd Edition)*. Springer, Berlin, Germany.
- HAREL 1987 Harel, D (1987): *Statecharts: A Visual Formulation for Complex Systems*. Science of Computer Programming 8(3): 231-274.
- HAUGH 1987 Haugh, H.A. (1987): *Non-Standard Semantics for the Method of Temporal Arguments*. Proceedings 10th International Joint Conference on Artificial Intelligence (IJCAI'87), Milan, Italy: 449-454.
- HAVEMANN 1994 Havemann, K.; Köppler, H.; Haag, U. (1994): *Integratives Konzept zur Behandlung Hoch-Maligner Non-Hodgkin-Lymphome. Studie A*. Marburg, Germany.
- HEINL ET AL. 1999 Heinl, P.; Horn, S.; Jablonksi, S.; Neeb, J.; Stein, K.; Teschke, M. (1999): *A Comprehensive Approach to Flexibility in Workflow Management Systems*. In: [GEORGAKOPOULOS ET AL. 1999 A]: 79-88.
- HEINLEIN 2000 Heinlein, C. (2000): *Workflow- und Prozesssynchronisation mit Interaktionsausdrücken und -graphen*. Department of Computer Science, University of Ulm, Germany.
- HEINLEIN 2001 Heinlein, C. (2001): *Workflow and Process Synchronization with Interaction Expressions and Graphs*. Proceedings 17th International Conference on Data Engineering (ICDE'01), Heidelberg, Germany: 243-252.
- HELLER 2000 Heller, B. (2000): *Telematic and Computer-Based Quality Management in a Communication Network for Malignant Lymphoma*. Project Description, University of Leipzig, Germany. (see also www.lymphome.de).
- HENSINGER ET AL. 2000 Hensinger, H.; Reichert, M.; Bauer, Th.; Strzeletz, Th.; Dadam, P. (2000): *ADEPT_{workflow}: Advanced Workflow Technology for the Efficient Support of Adaptive, Enterprise-wide Processes*. Proceedings of Software Demonstration Track of 7th International Conference on Extending Database Technology (EDBT'00), Konstanz, Germany: 29-30.
- HERRMANN & BAYER 1998 Herrmann, T.; Bayer, E. (1996): *Datenschutz und arbeitsrechtliche Aspekte beim Workflow-Management*. Proceedings of GI Chapter 2.5.2 (EMISA) Annual Meeting 1998 on Methods for Developing Information Systems and their Application, Gelsenkirchen, Germany: 125-136. (see also (<http://sunsite.informatik.rwth-aachen.de/Publications/CEUR-WS/Vol-15/>)).
- HERRTWICH & HOMMEL 1994 Herrtwich, R.G.; Hommel, G. (1994): *Nebenläufige Programme*. Springer, Berlin, Germany.
- HERRE & WAGNER 1996 Herre, H.; Wagner, G. (1997): *Stable Semantics for Temporal Deductive Databases*. Proceedings Post-Conference Workshop on Logic Programming and Deductive Databases of Joint International Conference and Symposium on Logic Programming (JICSLP'96), Bonn, Germany.

-
- HERTZBERG 1989 Hertzberg, J. (1989): *Planen – Einführung in die Planerstellungsmethoden der Künstlichen Intelligenz*. BI Wissenschaftsverlag, Mannheim, Germany.
- HOFFNER ET AL. 2000 Hoffner, Y; Ludwig, H; Gülcü, C; Grefen, P. (2000): *Architecture for Cross-Organisational Business Processes*. Proceedings 2nd International Workshop on Advanced issues of E-Commerce and Web-Based Information Systems (WECWIS'2000), Milpitas, CA, USA: 2-11.
- HORN & JABLONSKI 1998 Horn, S.; Jablonksi, S. (1999): *An Approach to Dynamic Instance Adaption in Workflow Management Applications*. Proceedings Workshop towards Adaptive Workflow Systems of ACM 1998 Conference on Computer Supported Cooperative Work (CSCW'98), Seattle, WA, USA.

[I]

- IABG 1996 IABG (1996): *Vorgangssteuerungssystem ProMInanD*. Handbücher, Version 2.0. Industrieanlagen-Betriebsgesellschaft mbH, Ottobrunn / Munich, Germany.
- IABG 2002 IABG (2002): *ProMInanD - Produktinformationen*. Industrieanlagen-Betriebsgesellschaft mbH, Ottobrunn / Munich, Germany. (<http://www.iabg.de/home-deutsch/information+kommu/it-realisation/index.html>).
- IBM 2002 A IBM (2002): *IBM MQSeries Workflow. Version 3.3*. IBM Company, New York, NY, USA.
- IBM 2002 B IBM (2002): *IBM MQSeries Workflow: Getting Started with Buildtime. Version 3.3*. IBM Company, New York, NY, USA. (available at: <http://www-3.ibm.com/software/ts/mqseries/library/manualsa/>).
- IBM 2002 C IBM (2002): *IBM MQ Series Workflow: Reference and Programming Guides. Version 3.3*. IBM Company, New York, NY, USA. (available at: <http://www-3.ibm.com/software/ts/mqseries/library/manualsa/>).
- IBM 2002 D IBM (2002): *MQSeries: An Introduction to Messaging and Queuing*. IBM Company, New York, NY, USA. (available at: <http://www-3.ibm.com/software/ts/mqseries/library/manualsa/>).
- I-LOGIXS 2002 I-Logixs Inc. (2002): *Statemate/Rhapsody White Papers*. I-Logixs Inc., Andover, MA, USA. (available at: http://www.ilogix.com/whitepapers_c.htm).
- INABA ET AL. 1998 Inaba, A.; Fujiwara, F.; Suzuki, T.; Okuma, S. (1998): *Timed Petri Net-Based Scheduling for Mechanical Assembly - Integration of Planning and Scheduling*. IEICE Transactions on Fundamentals in Electronics, Communications and Computer Science 81-A (4): 615-625.
- INGENERF 1993 Ingenerf, J. (1993): *Benutzeranpassbare Semantische Sprachanalyse und Begriffsrepräsentation für die Medizinische Dokumentation*. Dissertation, University of Aachen, Germany.
- IONA 2002 IONA Technologies PLC (2002): *Orbix 3.3 Documentation*. Dublin, Ireland. (available at: <http://www.orbixhome.com/docs/manuals/>).

[J]

- JABLONSKI 1997 Jablonski, S. (1997): *Architektur von Workflow-Management-Systemen*. Informatik Forschung und Entwicklung 12(2): 72-81.
- JABLONSKI ET AL. 1997 Jablonski, S.; Stein, K.; Teschke, M. (1997): *Experiences in Workflow Management for Scientific Computing*. Proceedings 8th International Workshop on Database and Expert Systems Applications (DEXA'97), Toulouse, France: 56-61.

-
-
- JAJODIA & KERSCHBERG 1997 Jajodia, S.; Kerschberg, L. (Eds.) (1997): *Advanced Transaction Models and Architectures*. Kluwer, New York, NY, USA.
- JARKE & OBERWEIS 1999 Jarke, M.; Oberweis, A. (Eds.) (1999): *Proceedings 11th International Conference on Advanced Information Systems Engineering (CAiSE'99), Heidelberg, Germany (LNCS 1626)*. Springer, Berlin, Germany.
- JENNINGS 2000 Jennings, N.R. (2000): *On Agent-based Software Engineering*. Artificial Intelligence 117: 277-296.
- JENNINGS ET AL. 2000 Jennings, N. R.; Faratin, P.; Norman, T. J.; O'Brien, P.; Odgers, B. (2000): *Autonomous Agents for Business Process Management*. International Journal of Applied Artificial Intelligence 14 (2): 145-189.
- JENSEN ET AL. 1996 Jensen, C. S.; Snodgrass, R. T.; Soo, M. D. (1996): *Extending Existing Dependency Theory to Temporal Databases*. IEEE Transactions on Knowledge and Data Engineering 8(4): 563-582.
- JEUSFELD ET AL. 1998 Jeusfeld, M.A.; Quix, C.; Jarke, M. (1998): *Design and Analysis of Quality Information for Data Warehouses*. Proceedings 17th International Conference on Conceptual Modeling (ER'98), Singapore (LNCS 1507). Springer, Berlin, Germany: 349-362.
- JÖDECKE 1997 Jödecke, E. (1997): *Konzept und Implementierung eines datenbankgestützten Dokumentenstruktur-Editors und -Generators für die Medizin (Concept and Implementation of a Document Type Editor and Generator for Medicine)*. Diploma thesis, University of Leipzig, Germany.
- JOERIS & HERZOG 1998 Joeris, G.; Herzog, O. (1998): *Managing Evolving Workflow Specifications*. Proceedings 3rd IFCIS International Conference on Cooperative Information Systems (CoopIS'98), New York, NY, USA, Aug. 1998: 310-319.
- JOERIS 1999 Joeris, G. (1999): *Defining Flexible Workflow Execution Behaviors*. In: Dadam, P.; Reichert, M. (Eds.): *Proceedings Workshop on Enterprise-Wide and Cross-Enterprise Workflow Management: Concepts, Systems, Applications at the Annual Conference of the German Computer Society, Paderborn, Germany (GI'99)*. Technical Report 99-07, Department of Computer Science, University of Ulm, Germany: 49-55.
- JOERIS & HERZOG 1999 Joeris, G.; Herzog, O. (1999): *Flexible and High-Level Modeling and Enacting of Processes*. In: [JARKE & OBERWEIS 1999]: 88-102.
- JONSSON & LIBERATORE 1999 Jonsson, P.; Liberatore, P. (1999): *On the Complexity of Finding Satisfiable Subinstances in Constraint Satisfaction*. Electronic Colloquium on Computational Complexity (ECCC) 6(038).
- [K]
- KAFEZA & KARLAPEM 1999 Kafeza, K.; Karlapalem, K (1999): *Temporally Constrained Workflows*. Proceedings 5th International Computer Science Conference (ICSC'99), Hong Kong, China (LNCS 1749). Springer, Berlin, Germany: 246-255.
- KAMATH 1998 Kamath, M. (1998): *Improving Correctness and Failure Handling in Workflow Management Systems*. PhD thesis, University of Massachusetts, USA.
- KAMATH & RAMAMRITHAM 1996 Kamath, M.; Ramamritham, K. (1996): *Correctness Issues in Workflow Management*. Distributed Systems Engineering Journal 3(4) (Special Issue on Workflow Management Systems): 213-221.

-
-
- KAMATH & RAMAMRITHAM 1998 Kamath, M.; Ramamritham, K. (1998): *Failure Handling and Coordinated Execution of Concurrent Workflows*. Proceedings 14th International Conference on Data Engineering (ICDE'98), Orlando, FL, USA: 334-341.
- KARBE ET AL. 1990 Karbe, B.; Ramsperger, N.; Weiss, P. (1990): *Support of Cooperative Work by Electronic Circulation Folders*. Proceedings ACM SIGOA Conference on Office Information Systems (OIS'90), New York, NY, USA: 109 - 117.
- KARK & KARL 2000 Karl, R.; Karl, S.; Becker, J.; Rosemann, M.; Heß, H. (2000): *DSK-Studie zu Workflow-Management und Dokument-Management*. dsk Beratungs-GmbH, Pfaffenhofen, Germany.
- KERZNER 2001 Kerzner, H. (2001): *Project Management: A Systems Approach to Planning, Scheduling, and Controlling*. Wiley, New York, NY, USA.
- KERAVNOU 1999 Keravnou, E.T. (1999): *A Multidimensional and Multigranular Model of Time for Medical Knowledge-Based Systems*. Journal of Intelligent Information Systems 13 (1/2): 73-120.
- KIEPUSZEWSKI ET AL. 2000 Kiepuszewski, B.; Hofstede, A.H.M. ter; Bussler, C. (2000): *On Structured Workflow Modeling*. Proceedings 12th International Conference on Advanced Information Systems Engineering (CAiSE'2000), Stockholm, Sweden (LNCS 1789). Springer, Berlin, Germany: 431-445.
- KIEPUSZEWSKI ET AL. 1998 Kiepuszewski, B.; Muhlberger, R.; Orlowska, M. (1998): *FlowBack: Providing Backward Recovery for Workflow Management Systems*. Proceedings ACM International Conference on Management of Data (SIGMOD'98), Seattle, WA, USA: 555-557.
- KIFER & LAUSEN 1989 Kifer, M.; Lausen, G. (1989): *F-Logic: A Higher-Order Language for Reasoning about Objects, Inheritance, and Scheme*. Proceedings ACM International Conference on Management of Data (SIGMOD'89), Portland, OR, USA: 134-146.
- KIFER & LOZINSKII 1992 Kifer, M.; Lozinskii, E.L. (1992): *A Logic for Reasoning with Inconsistency*. Journal of Automated Reasoning 9(2): 179-215.
- KIFER & SUBRAHMANIAN 1992 Kifer, M.; Subrahmanian, V.S. (1992): *Theory of Generalized Annotated Logic Programming and its Applications*. Journal of Logic Programming 12(4): 335-368.
- KIFER ET AL. 1995 Kifer, M.; Lausen, G.; Wu, J. (1995): *Logical Foundations of Object-Oriented and Frame-Based Languages*. Journal of the ACM, 42(4): 741-843.
- KLEIN & DELLAROCAS 1999 Klein M.; Dellarocas C. (1999): *Exception Handling in Agent Systems*. Proceedings 3rd Annual Conference on Autonomous Agents (AGENTS'99), Seattle, WA, USA: 62-68.
- KLEIN & DELLAROCAS 2000 Klein M.; Dellarocas C. (2000): *A Knowledge-Based Approach to Handling Exceptions in Workflow Systems*. Journal of Computer-Supported Collaborative Work. Special Issue on Adaptive Workflow Systems 9(3/4): 399-412.
- KLINGEMANN ET AL. 1999 Klingemann, K.; Wäsch, J.; Aberer, K. (1999): *Deriving Service Models in Cross-Organizational Workflows*. In: [RIDE 1999]: 100-107.
- KÖHLER 1998 Köhler, J. (1998): *Planning under Resource Constraints*. Proceedings 13th European Conference on Artificial Intelligence (ECAI'98), Brighton, UK: 489-493.
- KOBIELUS 2000 Kobielus, J.G. (2000): *BizTalk: Implementing Business-to-Business E-commerce*. Prentice Hall, Upper Saddle River, NJ, USA.

-
-
- KOKSAL ET AL. 1999 Koksall, P.; Cingil, I.; Dogac, A. (1999): *A Component-Based Workflow System with Dynamic Modifications*. Proceedings Next Generation Information Technologies and Systems (NGITS'99), Zikhron-Yaakov, Israel (LNCS 1649). Springer, Berlin, Germany: 238-255.
- KOLODNER ET AL. 1985 Kolodner, J.K.; Simpson, R.L. Jr.; Sycara-Cyranski, K. (1985): *A Process Model of Cased-Based Reasoning in Problem Solving*. Proceedings 9th International Joint Conference on Artificial Intelligence(IJCAI'85), Los Angeles, CA, USA: 284-290.
- KOUDAS ET AL. 2000 Koudas, N.; Indyk, P.; Muthukrishnan S. (2000): *Identifying Representative Trends in Massive Time Series Data Sets Using Sketches*. Proceedings 26th International Conference on Very Large Data Bases (VLDB'00), Cairo, Egypt: 363-372.
- KRADOLFER ET AL. 1999 Kradolfer, M.; Geppert, A.; Dittrich, K.R. (1999): *Workflow Specification in TRAMs*. Proceedings 18th International Conference on Conceptual Modeling (ER'99), Paris, France (LNCS 1728). Springer, Berlin, Germany: 263-277.
- KRADOLFER & GEPPERT 1999 Kradolfer, M.; Geppert, A. (1999): *Dynamic Workflow Schema Evolution based on Workflow Type Versioning and Workflow Migration*. Proceedings 4th International Conference on Cooperative Information Systems (CoopIS'99), Edinburgh, UK: 104-114.
- KRAUS 1997 Kraus, S. (1997): *Negotiation and Cooperation in Multi-Agent Environments*. Artificial Intelligence 94:79-97.
- KUBICEK & REICHERT 1996 Kubicek, M.; Reichert, M. (1996): *Das Workflow-Management-System ProMinanD*. Abschlußbericht Praktikum Workflow-Management-Systeme WS1995/96. Department for Computer Science, University of Ulm, Germany.
- KUHN ET AL. 1995 Kuhn, K.; Reichert, M.; Dadam, P. (1995): *Unterstützung der klinischen Kooperation durch Workflow-Management-Systeme*. Proceedings 40th German Medical Informatics Conference (GMDS'95), Bochum, Germany: 437-441.
- KULKARNI ET AL. 1999 Kulkarni, K.G.; Mattos, N.; Cochrane, R. (1999): *Active Database Features in SQL3*. In: [PATON 1999]: 197-219
- [L]
- LAKOS 1995 Lakos, C.A.: *From Coloured Petri Nets to Object Petri Nets*. Proceedings 16th International Conference on Application and Theory of Petri Nets (ICATPN'95), Turin, Italy (LNCS 935). Springer, Berlin, Germany: 278-297.
- LARSSON & HAYES-ROTH 1998 Larsson, J.E.; Hayes-Roth, B. (1998): *An Intelligent Autonomous Agent for Medical Monitoring and Diagnosis*. IEEE Intelligent Systems 13(1): 58-64.
- LAZCANO ET AL. 2000 Lazcano, A.; Alonso, G.; Schuldt, H.; Schuler, C. (2000): *The WISE Approach to Electronic Commerce*. International Journal of Computer Systems Science & Engineering (Special Issue on Flexible Workflow Technology Driving the Networked Economy) 15(5).
- LEANDER 1999 Leander, R. (1999): *Building Application Servers*. Cambridge University Press, Cambridge, UK.
- LEE & ELMASRI 1998 Lee, J.Y.; Elmasri, R. (1998): *An EER-Based Conceptual Model and Query Language for Time-Series Data*. Proceedings 17th International Conference on Conceptual Modeling (ER'98), Singapore (LNCS 1507). Springer, Berlin, Germany: 21-34.

-
-
- LESSER 1998 Lesser, V.R. (1998): *Reflections on the Nature of Multi-Agent Coordination and its Implications for an Agent Architecture*. Journal of Autonomous Agents and Multi-Agent Systems: 89-111.
- LEYMANN 1995 Leymann, F. (1995): *Supporting Business Transactions via Partial Backward Recovery in Workflow Management Systems*. Proceedings 6th Bi-Annual German Database Conference (BTW'95), Dresden, Germany: 51-70.
- LEYMANN 1997 Leymann, F. (1997): *Transaktionsunterstützung für Workflows*. Informatik Forschung und Entwicklung 12(2): 82-90.
- LEYMANN & ROLLER 2000 Leymann, F.; Roller, D. (2000): *Production Workflow: Concepts and Techniques*. Prentice Hall, Upper Saddle River, NJ, USA.
- LIEBHART 1998 Liebhart, W. (1998): *Fehler- und Ausnahmebehandlung im Workflow Management*. Dissertation, University of Klagenfurt, Austria.
- LIU & CONRADI 1993 Liu, C.; Conradi, R. (1993): *Automatic Replanning of Task Networks for Process Model Evolution in EPOS*. Proceedings 4th European Software Engineering Conference (ESEC'93), Garmisch-Partenkirchen, Germany: 434-450.
- LIU ET AL. 1998 A Liu, C.; Orlowska, M.E.; Li, H. (1998): *Automating Handover in Dynamic Workflow Environments*. Proceedings 10th International Conference on Advanced Information Systems Engineering (CAiSE'98), Pisa, Italy (LNCS 1789). Springer, Berlin, Germany: 159-171
- LIU ET AL. 1998 B Liu, L.; Yan, L.; Özsü, T. (1998): *Interoperability in Large-Scale Distributed Information Delivery Systems*. In: [DOGAC ET AL. 1998]: 246-280.
- LIU & PU 1998 A Liu, L.; Pu, C. (1998): *Methodical Restructuring of Complex Workflow Activities*. Proceedings 14th International Conference on Data Engineering (ICDE'98), Orlando, FL, USA: 342-350.
- LIU & PU 1997 Liu, L.; Pu, C. (1997): *ActivityFlow: Towards Incremental Specification and Flexible Coordination of Workflow Activities*. Proceedings 16th International Conference on Conceptual Modeling (ER'97), Los Angeles, CA, USA (LNCS 1331). Springer, Berlin, Germany: 169-182.
- LOPEZ ET AL. 1997 Lopez, C.; Sánchez, G.; Villegas, M. (1996): *An IDL-to-SQL mapping and how to automate its usage through C++ classes*. Workshop-Proceedings 11th European Conference on Object-Oriented Programming (ECOOP'97), Jyväskylä, Finland.
- LUDÄSCHER 1998 Ludäscher, B. (1998): *Integration of Active and Deductive Database Rules*. Dissertation, University of Freiburg, Germany.
- LUDWIG 1999 Ludwig, H. (1999): *Termination Handling in Inter-Organisational Workflows - An Exception Management Approach*. Proceedings 7th Euromicro Workshop on Parallel and Distributed Processing (PDP '99), Funchal, Portugal: 122 - 129.
- LUDWIG ET AL. 1999 Ludwig, H.; Grefen, P.; Bussler, C.; Shan, M.-C. (1999) (Eds.): *Proceedings WACC'99 Workshop on Cross-Organizational Workflows*. San Francisco, CA, USA.
- LUDWIG & HOFFNER 1999 Ludwig, H.; Hoffner, Y. (1999): *Contract-based Cross-Organisational Workflows - The Cross-Flow Project*. In: [LUDWIG ET AL. 1999].
- LUDWIG & WHITTINGHAM 1999 Ludwig, H.; Whittingham, K. (1999): *Virtual Enterprise Co-ordinator - Agreement-Driven Gateways for Cross-Organisational Workflow Management*. In: [GEORGAKOPOULOS ET AL. 1999 A]: 29-38.

[M]

- MA & KNIGHT 2001 Ma, J.; Knight, B. (2001): *Reified Temporal Logics: An Overview*. Artificial Intelligence Review 15(3): 189-217.
- MAIER 1986 Maier, D. (1986): *A Logic for Objects*. Proceedings 1986 Workshop on Foundations of Deductive Databases and Logic Programming, Washington D.C., USA: 6-26.
- MANN 1995 Mann, G. (1995): *An Object-Oriented Model for the Integration of Knowledge Based Systems*. Proceedings 8th World Congress on Medical Informatics (MEDINFO'95), Vancouver, Canada: 938-942.
- MANNA & PNUELI 1992 Manna, Z.; Pnueli, A. (1992): *The Temporal Logic of Reactive and Concurrent Systems*. Springer, Berlin, Germany.
- MARCHAL 1999 Marchal, B. (1999): *XML by Example*. Que, Indianapolis, IN, USA.
- MARJANOVIC & ORLOWSKA 1999 Marjanovic, O.; Orłowska, M.E. (1999): *On Modeling and Verification of Temporal Constraints in Production Workflows*. Knowledge and Information Systems 1(2): 157-192.
- MAY 1999 May, W. (1999): *A Tableau Calculus for a Temporal Logic with Temporal Connectives*. Proceedings Automated Reasoning with Analytic Tableaux and Related Methods, (TABLEAUX'99), Albany, NY, USA (LNCS 1617). Springer, Berlin, Germany: 232-246.
- MAY ET AL. 1997 May, W.; Schlepphorst, C.; Lausen, G. (1997): *Integrating Dynamic Aspects into Deductive Object-Oriented Databases*. Proceedings 3rd International Workshop on Rules in Database Systems (RIDS'97), Skövde, Sweden (LNCS 1312). Springer, Berlin, Germany: 20-34.
- MCBRIEN & POULOVASSILIS 1998 McBrien, P.; Poulouvassilis, A. (1998): *A Formalisation of Semantic Schema Integration*. Information Systems 23(5): 307-334.
- MCDERMOTT 1982 McDermott, D. (1982): *A Temporal Logic for Reasoning About Processes and Plans*. Cognitive Science 6: 101-155.
- MCDERMOTT & HENDLER 1995 McDermott, D.; Hendler, J.A. (1995): *Planning: What it is, What it could be, An Introduction to the Special Issue on Planning and Scheduling*. Artificial Intelligence 76(1-2): 1-16.
- MERZ ET AL. 1996 Merz, M.; Liberman, B.; Müller-Jones, K.; Lamersdorf, W. (1996): *Inter-Organisational Workflow Management with Mobile Agents in COSM*. Proceedings 1st International Conference on the Practical Application of Intelligent Agents and Multi-Agent Technology (PAAM'96), London, UK: 405-420.
- MESEGUER 1992 Meseguer, P. (1992): *Incremental Verification of Rule-Based Expert Systems*. Proceedings 10th European Conference on Artificial Intelligence (ECAI'92), Vienna, Austria: 840-844.
- MEYER & SCHOBENS 1999 Meyer, J.-J.C.; Schobbens, P.-Y. (Eds.) (1999): *Formal Models of Agents (LNCS 1760)*. Springer, Berlin, Germany.
- MICHAELIS 1997 Michaelis, J. (1997): *Biostatistical Methods*. In: [BEMMEL & MUSEN 1997]: 387-397.
- MILLER ET AL. 1998 Miller, J.A.; Palaniswami, D.; Sheth, A.; Kochut, K.; Singh, H. (1998): *WebWork: METEOR's Web-Based Workflow Management System*. Journal of Intelligent Information Systems 10(2): 185-215.

-
-
- MOHAN 1996 Mohan, C. (1996): *State of the Art in Workflow Management Research and Products (Tutorial)*. Proceedings ACM International Conference on Management of Data (SIGMOD'96), Montreal, Canada: 544.
- MONSON-HAEFEL 2000 Monson-Haefel, R. (2000): *Enterprise JavaBeans*. O'Reilly, Farnham, UK.
- MORICE ET AL. 1995 Morice, V.; Séroussi, B.; Boisvieux, J.F. (1995): *A Real Time Control Architecture for Continuously Managing Patients in a Care Unit*. Methods of Information in Medicine 34(5): 475-88.
- MOSTERMAN & BISWAS 1997 Mosterman, P.; Biswas, G. (1997): *Monitoring, Prediction, and Fault Isolation in Dynamic Physical Systems*. Proceedings 14th National Conference on Artificial Intelligence and 9th Innovative Applications of Artificial Intelligence Conference (AAAI'97, IAAI'97), Providence, RI, USA: 100-105.
- MOTAKIS & ZANIOLO 1997 A Motakis, I.; Zaniolo, C. (1997): *Temporal Aggregation in Active Database Rule*. Proceedings ACM International Conference on Management of Data (SIGMOD'97), Tucson, AZ, USA: 440-451.
- MOTAKIS & ZANIOLO 1997 B Motakis, I. Zaniolo, C. (1997): *Formal Semantics for Composite Temporal Events in Active Database Rules*. Journal of System Integration 7(3-4): 291-325.
- MÜLLER 1994 Müller, R. (1994): *Ein Patienten-Datenmodell für die Kinderonkologie (a Patient Data Model for Pediatric Oncology)*. Diploma thesis, University of Mainz, Germany.
- MÜLLER 1996 Müller, H.J. (1996): *Negotiation Principles*. In: O'Hare, G.M.P.; Jennings, N.R. (Eds.): *Foundations of Distributed Artificial Intelligence*. Wiley, New York, NY, USA.
- MÜLLER 1997 Müller, R. (1997): *The CliniCon Framework for Context Representation in Electronic Patient Records*. Proceedings American Medical Informatics Association Annual Fall Symposium 1997, Nashville, TN, USA: 178-182.
- MÜLLER ET AL. 1997 A Müller, R.; Thews, O.; Rohrbach, C.; Sergl, M.; Pommerening, K. (1997): *A Graph-Grammar Approach to Represent Causal, Temporal and Other Contexts in an Oncological Patient Record*. Yearbook of Medical Informatics 1997 of the International Medical Informatics Association. Schattauer, Stuttgart, Germany: 230-244.
- MÜLLER ET AL. 1997 B Müller, R.; Sergl, M.; Nauerth, U.; Dittrich, H.M.; Schoppe, D.; Pommerening, K. (1997): *TheMPO: A Knowledge-Based System for Therapy Planning in Pediatric Oncology*. Computers in Biology and Medicine 27(3): 177-200.
- MÜLLER ET AL. 1997 C Müller, J.P.; Wooldridge, M.J.; Jennings, N.R. (Eds.) (1997): *Intelligent Agents III*. Proceedings 4th International Workshop on Agent Theories, Architectures, and Languages (ATAL'96), Budapest, Hungary (LNCS 1193). Springer, Berlin, Germany.
- MÜLLER ET AL. 1998 Müller, R.; Heller, B.; Löffler, M.; Rahm, E.; Winter, A. (1998): *HematoWork: A Knowledge-based Workflow System for Distributed Cancer Therapy*. Proceedings 43rd German Medical Informatics Conference (GMDS'98), Bremen, Germany: 63-66.
- MÜLLER & HELLER 1998 Müller, R.; Heller, B. (1998): *A Petri Net-based Model for Knowledge-based Workflows in Distributed Cancer Therapy*. In: [BUKHRES ET AL. 1998]: 91-99.
- MÜLLER ET AL. 1999 A Müller, R.; Stöhr, T.; Rahm, E. (1999): *An Integrative and Uniform Model for Metadata Management in Data Warehousing Environments*. Proceedings Workshop on Design and Management of Data Warehouses (DMDW'99), Heidelberg, Germany.

-
-
- MÜLLER ET AL. 1999 B Müller, J.P.; Singh, M.P.; Rao, A.S. (Eds.) (1999): *Intelligent Agents V*. Proceedings 5th International Workshop on Agent Theories, Architectures, and Languages (ATAL'98), Paris, France (LNCS 1555). Springer, Berlin, Germany.
- MÜLLER & RAHM 1999 Müller, R.; Rahm, E. (1999): *Rule-Based Dynamic Modification of Workflows in a Medical Domain*. Proceedings 8th Bi-Annual German Database Conference (BTW'99), Freiburg, Germany: 429-448.
- MÜLLER & RAHM 2000 Müller, R.; Rahm, E. (2000): *Dealing with Logical Failures for Collaborating Workflows*. Proceedings 5th International Conference on Cooperative Information Systems (CoopIS'2000), Eilat, Israel (LNCS 1901). Springer, Berlin, Germany: 210-223.
- MURATA 1984 Murata, T. (1984): *Petri Nets and Their Applications: An Introduction*. In: Chang, S. K. (Ed.): *Management and Office Information Systems*. Plenum Press, New York, NY, USA: 351-367.
- MUSEN ET AL. 1996 Musen, M.A.; Tu, S.W.; Das, A.K.; Shahar, Y. (1996): *EON: A Component-Based Architecture for Automation of Protocol-Directed Therapy*. Journal of the American Medical Informatics Association 3(6): 367-388.
- MUTH ET AL. 1998 A Muth, P.; Wodtke, D.; Weißenfels, J.; Kotz-Dittrich, A.; Weikum, G. (1998): *From Centralized Workflow Specification to Distributed Workflow Execution*. Journal of Intelligent Information Systems 10(2): 159-184.
- MUTH ET AL. 1998 B Muth, P.; Wodtke, D.; Weißenfels, J.; Weikum, G.; Kotz-Dittrich, A. (1998): *Enterprise-Wide Workflow Management based on State and Activitycharts*. In: [DOGAC ET AL. 1998]: 281-303.
- MYERS 1997 Myers, K.L. (1997): *Abductive Completion of Plan Sketches*. Proceedings 14th National Conference on Artificial Intelligence and 9th Innovative Applications of Artificial Intelligence Conference (AAAI'97, IAAI'97), Providence, RI, USA: 687-693.
- MYERS 1998 Myers, K. (1998): *Towards a Framework for Continuous Planning and Execution*. Proceedings AAAI Symposium on Distributed Continual Planning, Menlo Park, CA, USA.
- [N]
- NAYLOR 1995 Naylor, H.F.W. (1995): *Construction Project Management: Planning and Scheduling*. Delmar Publishers, Clifton Park, NY, USA.
- NEUMANN & MORLOCK 1993 Neumann, K.; Morlock, M. (1993): *Operations Research*. Carl Hanser, Munich, Germany.
- NGU 1999 Ngu, A.H.H. (1999): *Specification of Cooperative Constraints in Virtual Enterprise Workflow*. In: [RIDE 1999]: 140-147.
- NOVOPACHENNYI ET AL. 1997 Novopachennyi, I.; Oguro, Y.; Wischnewsky, M. B.; Zhao, J. (1997): *Wissensbasierte Systeme in der Onkologie am Beispiel von ONCO-CONS*. Künstliche Intelligenz 97 (3): 47-51.
- NEUBERT 1999 Neubert, U. (1999): *CORBA-Integration des Workflow-Management-Systems IBM FlowMark. (CORBA Integration of the Workflow Management System IBM Flowmark)*. Diploma thesis, University of Leipzig, Germany.

[O]

- OBERWEIS ET AL. 1994 Oberweis, A.; Scherrer, G.; Stucky, W. (1994): *INCOME/STAR: Methodology and Tools for the Development of Distributed Information Systems*. Information Systems 19(8): 643-682.

-
-
- OBERWEIS 1996 Oberweis, A. (1996): *Modellierung und Ausführung von Workflows mit Petri-Netzen*. Teubner, Berlin, Germany.
- OMG 2002 A Object Management Group (2002): *CORBA services - Event Management Service*. <http://www.omg.org>.
- OMG 2002 B Object Management Group (2002): *CORBA services - Transaction Service*. <http://www.omg.org>.
- OMG 1998 Object Management Group; Business Object Domain Task Force (BODTF), Workflow Workgroup (1998): *Final Standard of Workflow Management Facility Specification*. <http://www.db.inf.tu-dresden.de/dokumente/wf-wg/documents.html>.
- ORFALI ET AL. 1995 Orfali, R.; Harkey, D.; Edwards, K. (1995): *The Essential Distributed Objects Survival Guide*. Wiley, New York, NY, USA.
- ÖZSOYOGLU & SNODGRASS 1995 Özsoyoglu, G.; Snodgrass, R.T. (1995): *Temporal and Real-Time Databases: A Survey*. IEEE Transactions on Knowledge and Data Engineering 7(4) (Special Section on Temporal and Real-Time Databases): 513-532.
- O₂ 1998 O₂ (1998): *O₂ CORBA User Manual*.

[P]

- PANAGOS & RABINOVICH 1996 Panagos, E.; Rabinovich, M. (1996): *Escalations in Workflow Management Systems*. Proceedings Workshop on Databases: Active & Real-Time (DART'96), Rockville, MD, USA.
- PAPAZOGLU & TSALGATIDOU 1999 Papazoglou, M.P.; Tsalgatiidou, A. (1999): *Guest Editorial: Special Issue on Information Systems Support for Electronic Commerce*. Information Systems 24(6): 425-427.
- PATON 1999 Paton, N.W. (Ed.) (1999): *Active Rules in Database Systems*. Springer, New York, NY, USA.
- PERRAJU & PRASAD 2000 Perraju, S.P.; Prasad, B.E. (2000): *An Algorithm for Maintaining Working Memory Consistency in Multiple Rule Firing Systems*. Data & Knowledge Engineering 32(2): 181-198.
- PETRI 1962 Petri, C.A. (1962): *Kommunikation mit Automaten*. Dissertation, Technical University of Darmstadt, Germany.
- PFREUNDSCHUH & LÖFLER 1994 Pfreundschuh, M.; Löffler, M. (1994): *Integratives Konzept zur Behandlung hochmaligner Non-Hodgkin-Lymphome*. Studie B. Homburg and Leipzig, Germany.
- PFREUNDSCHUH & TRÜMPER 1998 Pfreundschuh, M.; Trümper, L. (1998): *Behandlungsstrategien bei hochmalignen Non-Hodgkin-Lymphomen*. Schweizerische Rundschau für Medizin, Praxis 87(23): 812-815.
- PHILIPOSE 1986 Philipose, S. (1986): *Operations Research - A Practical Approach*. Tata McGraw-Hill, New York, NY, USA.
- POCOCK 1993 Pocock, S.J. (1993): *Clinical Trials: A Practical Approach*. Wiley, New York, NY, USA.
- POECK 1995 Poeck, K. (1995): *Konfigurierbare Problemlösungsmethoden am Beispiel der Problemklassen Zuordnung und Diagnostik*. Dissertationen zur Künstlichen Intelligenz 86. infix, St. Augustin, Germany.

-
- POECK & PUPPE 1992 Poeck, K.; Puppe, F. (1992): *Coke: Efficient Solving of Complex Assignment Problems with the Propose-and-Exchange Method*. Proceedings 5th International Conference on Tools with Artificial Intelligence, Arlington, VA, USA: 136-143.
- POOLE ET AL. 1998 Poole, D.; Mackworth, A.; Goebel, R. (1998): *Computational Intelligence, A Logical Approach*. Oxford University Press, Oxford, UK.
- PRADHAN & VAIDYA 1994 B Dhiraj K. Pradhan, Nitin H. Vaidya (1994): *Roll-Forward Checkpointing Scheme: A Novel Fault-Tolerant Architecture*. IEEE Transactions on Computers 43(10): 1163-1174.
- PRIOR 1955 Prior (1955): *Diodoram Modalities*. Philosophical Quaterly 5: 205-213.
- PSCHYREMBEL 1994 Pschyrembel, W.; Hildebrandt, H. (1994): *Klinisches Wörterbuch*. Walter de Gruyter, Berlin, Germany.
- PUPPE 1993 Puppe, F. (1993): *Systematic Introduction to Expert Systems. Knowledge Representations and Problem-Solving Methods*. Springer, Berlin, Germany.
- PYARALI ET AL. 1996 Pyarali, I; Harrison, T.H.; Schmidt, D.C. (1996): *Design and Performance of an Object-Oriented Framework for High-Speed Electronic Medical Imaging*. USENIX Computing Systems Journal 9(4): 331-375.

[Q]

- QUAGLINI ET AL. 1999 Quaglini, S.; Mossa, C.; Fassino, C.; Stefanelli, M.; Cavallini, A.; Micieli, G. (1999): *Guidelines-Based Workflow Systems*. Proceedings Joint European Conference on Artificial Intelligence in Medicine and Medical Decision Making (AIMDM'99), Aalborg, Denmark (LNCS 1620). Springer, Berlin, Germany: 65-75.

[R]

- RAHM 1994 Rahm, E. (1994): *Mehrrechner-Datenbanksysteme. Grundlagen der verteilten und parallelen Datenbankverarbeitung*. Addison Wesley, Bonn, Germany.
- REICHERT & DADAM 1997 Reichert, M.; Dadam, P. (1997): *A Framework for Dynamic Changes in Workflow Management Systems*. Proceedings 8th International Workshop on Database and Expert Systems Applications (DEXA'97), Toulouse, France: 42-48.
- REICHERT & DADAM 1998 Reichert, M.; Dadam, P. (1998): *ADEPT_{FLEX} - Supporting Dynamic Changes of Workflows Without Losing Control*. Journal of Intelligent Information Systems 10(2): 93-129.
- REICHERT 2000 Reichert, M. (2000): *Dynamische Ablaufänderungen in Workflow-Management-Systemen*. Dissertation, University of Ulm, Germany.
- REICHERT & DADAM 2000 Reichert, M.; Dadam, P. (2000): *Geschäftsprozessmodellierung und Workflow-Management - Konzepte, Systeme und deren Anwendung*. Industrie Management 16(3): 23-27.
- REICHERT ET AL. 2000 Reichert, M.; Dadam, P.; Mangold, R.; Kreienberg, R. (2000): *Computerbasierte Unterstützung von Arbeitsabläufen im Krankenhaus - Konzepte, Technologien und deren Anwendung*. Zentralblatt für Gynäkologie 122: 53-67.
- REICHERT ET AL. 1998 Reichert, M.; Hensinger, C.; Dadam, P. (1998): *Supporting Adaptive Workflows in Advanced Application Environments*. In: [BUKHRES ET AL. 1998]: 100 - 109.

-
-
- REINWALD 1993 Reinwald, B. (1993): *Workflow-Management in verteilten Systemen*. Teubner, Stuttgart, Germany.
- REINWALD & MOHAN 1996 Reinwald, B.; Mohan, C. (1996): *Structured Workflow Management with Lotus Notes Release 4*. Proceedings 41st IEEE Computer Society International Conference (COMPCON '96), Santa Clara, CA, USA: 451-457.
- REITER 2001 Reiter, R. (2001): *Knowledge in Action: Logical Foundations for Specifying and Implementing Dynamical Systems*. MIT Press, Cambridge, MA, USA.
- REITER ET AL. 1992 Reiter, A.; Schrappe, M.; Ludwig, W.D.; Lampert, F.; Harbott, J.; Henze, G.; Niemeyer, C.; Gadner, H.; Müller-Weihrich, St.; Ritter, J.; Odenwald, E.; Riehm, H. (1992): *Favorable Outcome of B-Cell Acute Lymphoblastic Leukemia in Childhood: A Report of Three Consecutive Studies of the BFM Group*. Blood 80: 2471-2478.
- REITER 1984 Reiter, R. (1984): *Towards a Logical Reconstruction of Relational Database Theory*. In: Brodie, M.; Mylopoulos, J.; Schmidt, J. (Eds.): *On Conceptual Modeling*. Springer, Berlin, Germany: 191-233.
- REMEDY CORP. 2000 Remedy Corporation (2000): *Action Request System 4.0 Reference Manuals*. Remedy Corporation (<http://www.remedy.com>).
- REUTER & SCHWENKREIS 1995 Reuter, A.; Schwenkreis, F. (1995): *ConTracts - A Low-Level Mechanism for Building General-Purpose Workflow Management Systems*. IEEE Data Engineering Bulletin 18(1): 4-10.
- REUTER ET AL. 1997 Reuter, A.; Schneider, K.; Schwenkreis, F. (1997): *ConTracts Revisited*. In: [JAJODIA & KERSCHBERG 1997]: 127-151.
- RIEHM 1995 Riehm, H. (Studienleitung) (1995): *ALL-BFM 95: Therapiestudie zur Behandlung von Kindern und Jugendlichen mit Akuter Lymphoblastischer Leukämie (Therapieprotokoll)*. Medical University of Hannover, Germany.
- RIDE 1999 Proceedings 9th International Workshop on Research Issues on Data Engineering: Information Technology for Virtual Enterprises (RIDE'99), Sydney, Australia. IEEE Computer Society.
- ROANES-LOZANO ET AL. 2000 Roanes-Lozano, E.; Laita, L. M.; Roanes-Macas, E.; Maojo, V.; Corredor, V.; de la Vega, A.; Zamora, A. (2000): *A Groebner Bases-Based Shell for Rule-Based Expert Systems Development*. Expert Systems with Applications 18 (3): 221-230.
- ROZ98 Rozenberg, G. (1998): *Handbook of Graph Grammars and Computing by Graph Transformations*. (Volume 1: Foundations). World Scientific Publishing, Singapore.
- RUSINKIEWICZ & BREGOLIN 1997 Rusinkiewicz, M.; Bregolin, M. (1997): *Transactional Workflows in Distributed Systems*. Fundamenta Informaticae 30(3/4): 325-344.
- [S]
- SAMPAIO & PATON 2000 Sampaio, P.R.F.; Paton, N.W. (2000): *Query Processing in DOQL: A Deductive Database Language for the ODMG Model*. Data & Knowledge Engineering 35(1): 1-38.
- SANTUCCI 1998 Santucci, G. (1998): *Semantic Schema Refinements for Multilevel Schema Integration*. Data & Knowledge Engineering 25(3): 301-326.
- SARIN 1996 Sarin, S.K. (1996): *Workflow and Data Management in InConcert*. Proceedings 12th International Conference on Data Engineering (ICDE '96), New Orleans, LA, USA: 497-499.

-
-
- SAUNDERS 1999 Saunders, A. (1999): *Credit Risk Measurement: New Approaches to Value at Risk and Other Paradigms*. Wiley, New York, NY, USA.
- SCHEER 1998 A Scheer, A.-W (1998): *ARIS – Modellierungsmethoden, Metamodelle, Anwendungen*. Springer, Berlin, Germany.
- SCHMITT & TÜRKEK 1998 Schmitt, I.; Türker, C. (1998): *An Incremental Approach to Schema Integration by Refining Extensional Relationships*. Proceedings 7th ACM International Conference on Information and Knowledge Management (CIKM'98), Bethesda, MD, USA. ACM Press, New York, NY, USA: 322-330.
- SCHÖF ET AL. 1995 Schöf, S.; Sonnenschein, M.; Wieting, R. (1995): *High-level Modeling with THORNs*. Proceedings 14th International Congress on Cybernetics, Namur, Belgium: 453-458.
- SCHÖNING 1989 Schöning, U. (1998): *Logic for Computer Scientists*. Birkhäuser, Berlin, Germany.
- SCHMOOR ET AL. 1997 Schmoor, C.; Eisele, C.; Graf, E.; Sauerbrei, W.; Klingele, B.; Hellmer, A.; Rossner, R.; Schumacher, M. (1997): *Arbeitsweisen des Methodischen Zentrums am Institut für Medizinische Biometrie und Medizinische Informatik der Universität Freiburg bei der biometrischen Betreuung klinischer Studien*. Informatik, Biometrie und Epidemiologie in Medizin und Biologie 28 (4): 253-274.
- SCHULDT ET AL. 1999 Schuldt, H.; Alonso, G.; Schek, H.-J. (1999): *Concurrency Control and Recovery in Transactional Process Management*. Proceedings 18th ACM Symposium on Principles of Database Systems (PODS'99), Philadelphia, PA, USA.
- SCHULZE 1999 Schulze, W. (1999): *Workflow-Management für CORBA-basierte Anwendungen. Systematischer Architekturentwurf eines OMG-konformen Workflow-Management-Dienstes*. Springer, Berlin, Germany.
- SCHULZE ET AL. 1998 Schulze, W.; Bussler, Ch.; Meyer-Wegener, K. (1998): *Standardising on Workflow-Management - The OMG Workflow Management Facility*. ACM SIGGROUP Bulletin 19(3).
- SCHWALB & DECHTER 1997 Schwalb, E.; Dechter, R. (1997): *Processing Disjunctions in Temporal Constraint Networks*. Artificial Intelligence 93: 29-61.
- SCHWALB & VILA 1998 Schwalb, E.; Vila, L. (1998): *Temporal Constraints: A Survey*. Constraints 3(2/3): 129-149.
- SELLENTIN 1999 Sellentin, J. (1999): *Konzepte und Techniken der Datenversorgung für komponentenbasierte Informationssysteme*. Dissertation, University of Stuttgart, Germany.
- SELLENTIN 2000 Sellentin, J. (2000): *Konzepte und Techniken der Datenversorgung für Informationssysteme*. Informatik Forschung und Entwicklung 15(2): 92-109.
- SEMA GROUP 2000 Sema Group (2000): *The FORO Workflow System*. <http://dis.sema.es/projects/FORO/foro.html>.
- SENKUL ET AL. 2002 Senkul, P.; Kifer, M.; Toroslu, I.H. (2002): *A Logical Framework for Scheduling Work ows Under Resource Allocation Constraints*. Proceedings 28th International Conference on Very Large Databases (VLDB'02), Hong Kong, China.
- SESSIONS 1998 Sessions, R. (1998): *COM and DCOM: Microsoft's Vision for Distributed Objects*. Wiley, New York, NY, USA.
- SHAHAR & MUSEN 1996 Shahar, Y.; Musen, M (1996): *Knowledge-Based Temporal Abstraction in Clinical Domains*. Artificial Intelligence in Medicine 8(3): 267-298.

-
-
- SHAN ET AL. 1997 Shan, M.-Ch.; Davis, J.; Du, W.; Huang, Y. (1997): *HP Workflow Research: Past, Present, and Future*. In: [DOGAC ET AL. 1998]: 91-105.
- SHETH 1997 Sheth, A. (1997): *From Contemporary Workflow Process Automation to Adaptive and Dynamic Work Activity Coordination and Collaboration*. Proceedings 8th International Workshop on Database and Expert Systems Applications (DEXA'97), Toulouse, France: 24-27.
- SHETH & KOCHUT 1998 Sheth, A.; Kochut, K. (1998): *Workflow Applications to Research Agenda: Scalable and Dynamic Work Coordination and Collaboration Systems*. In: [DKÖ+98]: 35-60.
- SINGH & HUHN 1994 Singh, M.P.; Huhn, M.N. (1994): *Automating Workflows for Service Order Processing. Integrating AI and Database Technologies*. IEEE Expert 9(5): 19-23.
- SISTLA & WOLFSON 1995 Sistla, A.P.; Wolfson, O. (1995): *Temporal Triggers in Active Databases*. IEEE Transactions on Knowledge and Data Engineering 7(3): 471-486.
- SMITH & KANDEL 1993 Smith, S.; Kandel, A. (1993): *Verification and Validation of Rule-Based Expert Systems*. CRC Press, Boca Raton, FL, USA.
- SNODGRASS 1999 Snodgrass, R.T. (1999): *Developing Time-Oriented Database Applications in SQL*. Morgan Kaufmann, San Francisco, CA, USA.
- SNODGRASS ET AL. 1995 Snodgrass, R.T. (Ed.) (1995): *The TSQL2 Temporal Query Language*. Kluwer, New York, NY, USA.
- SNODGRASS ET AL. 1998 Snodgrass, R.T.; Böhlen, M.H.; Jensen, C.S.; Steiner, A. (1998): *Transitioning Temporal Support in TSQL2 to SQL3*. In: [ETZION ET AL. 1998]: 150-194.
- SOFTWARE LEY GMBH 2000 Software Ley GmbH (2000): *Cosa Reference Manuals*. Software-Ley GmbH, Pulheim, Germany (http://www.ley.de/_englisch/cosa/index.htm).
- SOLEY & KENT 1995 Soley, R.M.; Kent, W. (1995): *The OMG Object Model*. In: Kim, W. (Ed.): *Modern Database Systems: The Object Model, Interoperability, and Beyond*. Addison Wesley, Reading, MA, USA: 18-41.
- SON ET AL. 2001 Son, J.H.; Kim, J.H.; Kim, M.H. (2001): *Deadline Allocation in a Time-Constrained Workflow*. International Journal of Cooperative Information Systems 10(4): 509-530.
- SON & KIM 2001 Son, J.H.; Kim, M.H. (2001): *Improving the Performance of Time-constrained Workflow Processing*. Journal of Systems and Software 58 (3): 211-219.
- SOWA 1984 Sowa, J.F. (1984): *Conceptual Structures*. Addison Wesley, Reading, MA.
- SPEER 1997 Speer, R. (1997): *Modellierung und Realisierung einer Client/Server-Datenbank zur Erfassung und Auswertung medizinischer Studien (Model and Realization of a Client/Server Database for the Acquisition and Analysis of Clinical Trials)*. Diploma thesis, University of Leipzig, Germany.
- SPEER & HELLER 1998 Speer, R.; Heller, B. (1998): *Einsatz moderner Kommunikationstechniken bei der Durchführung multizentrischer klinischer Studien*. Proceedings 43rd German Medical Informatics Conference (GMDS'98), Bremen, Germany: 125-129.
- SQL/PSM 1996 ANSI/ISO Development Committees (1996): *SQL Standard Part 4: SQL/PSM (Persistent Storage Modules)*. (http://www.jcc.com/SQLPages/jccs_sql.htm)

-
- STARK & LACHAL 1995 Stark, H.; Lachal, L. (1995): *Ovum Evaluates: Workflow*. Ovum, London, UK.
- STURM & WOLTER 2002 Sturm, H.; Wolter, F. (2002): *A Tableau Calculus for Temporal Description Logic: The Expanding Domain Case*. To appear in: Journal of Logic and Computation.
- SUWA ET AL. 1982 Suwa, M.; Scott, A.C.; Shortliffe, E.H. (1982): *An Approach to Verifying Completeness and Consistency in a Rule-Based Expert System*. AI Magazine 3: 16-21.

[T]

- TABOADA 1996 Taboada, M.; Marín, R.; Mira, J.; Otero, R.P. (1996): *Integrating Medical Expert Systems, Patient Data-Bases and User Interfaces*. Journal of Intelligent Information Systems 7 (3): 261-285.
- TAHA 1982 Taha, H.A. (1982): *Operations Research*. Macmillan Publishing Company, Basingstoke, UK.
- TANENBAUM & WOODHULL 1997 Tanenbaum, A.S.; Woodhull, A.S. (1997): *Operating Systems: Design and Implementation*. Prentice Hall, Upper Saddle River, NJ, USA.
- TANG & VEIJALAINEN 1995 Tang, J.; Veijalainen, J. (1995): *Transaction-Oriented Workflow Concepts in Inter-Organizational Environments*. Proceedings 4th ACM International Conference on Information and Knowledge Management (CIKM'95), Baltimore, MD, USA: 250 - 259.
- TANSEL ET AL. 1993 Tansel, A.U.; Clifford, J.; Gadia S.K.; Jajodia, S.; Segev, A.; Snodgrass, R.T. (Eds.) (1993): *Temporal Databases: Theory, Design, and Implementation*. Benjamin Cummings, Menlo Park, CA, USA.
- THOMPSON & WATKINS 1997 Thompson, D.; Watkins, D.: (1997): *Comparisons between CORBA and DCOM: Architectures for Distributed Computing*. Technical report SD TR97-7. Department of Software Development, Monash University, Australia.
- TOMII ET AL. 1999 Tomii, N.; Zhou, L.; Fukumara, N. (1999): *An Algorithm for Station Shunting Scheduling Problems Combining Probabilistic Local Search and PERT*. Proceedings 12th International Conference on Industrial and Engineering Applications of Artificial Intelligence and Expert Systems (IEA/AIE'99), Cairo, Egypt (LNCS 1611). Springer, Berlin, Germany: 788-797.
- TU ET AL. 1989 Tu, S.W.; Kahn, M.G.; Musen, M.A.; Ferguson, J.C.; Shortliffe, E.H.; Fagan, L.M. (1989): *Episodic Skeletal-Plan Refinement Based on Temporal Data*. Communications of the ACM 32(12): 1439-1455.
- TU ET AL. 1995 Tu, S.W.; Eriksson, H.; Gennari, J.; Shahar, Y.; Musen, M. (1995): *Ontology-Based Configuration of Problem-Solving Methods and Generation of Knowledge-Acquisition Tools: Application of PROTEGE-II to Protocol-Based Decision Support*. Artificial Intelligence in Medicine 7: 257-289.

[V]

- VAN BEEK & MANCHAK 1996 van Beek, P.; Manchak, D.W. (1996): *The Design and Experimental Analysis of Algorithms for Temporal Reasoning*. Journal of Artificial Intelligence Research 4: 1-18.
- VERMEULEN ET AL. 2000 Vermeulen, V.; Bauwens, B.; Westerhuis, F.; Broos, R. (2000): *XML and CORBA, Synergistic or Competitive?* Proceedings 7th International Conference on Intelligence and Services in Networks (IS&N2000), Athens, Greece (LNCS 1774). Springer, Berlin, Germany: 155-168.

-
- VILA 1994 Vila, L. (1994): *A Survey on Temporal Reasoning in Artificial Intelligence*. Artificial Intelligence Communications 7(1): 4-28.
- VOORHOEVE & AALST 1997 Voorhoeve, M.; van der Aalst, W.: *Ad-hoc Workflow: Problems and Solutions*. Proceedings 8th International Workshop on Database and Expert Systems Applications (DEXA'97), Toulouse, France: 36-41.
- VOSSSEN & BECKER 1996 Vossen, G.; Becker, J. (Eds.) (1996): *Geschäftsprozeßmodellierung und Workflow-Management: Modelle, Methoden, Werkzeuge*. Thomson, Bonn, Germany.
- VOSSSEN & WESKE 1998 Vossen, G.; Weske, M. (1998): *The WASA Approach to Workflow Management for Scientific Applications*. In: [DOGAC ET AL. 1998]: 145-164.

[W]

- WÄCHTER 1996 Wächter, H. (1996): *Fehlertolerantes Workflow-Management*. Dissertation, University of Stuttgart, Germany.
- WÄCHTER & REUTER 1992 Wächter, H.; Reuter, A. (1992): *The ConTract Model*. In: [ELMARGARMID 1992]: 219-263.
- WÄSCH ET AL. 1998 Wäsch, J.; Aberer, K.; Neuhold, E.J. (1998): *Transactional Support for Cooperative Applications*. In: [DOGAC ET AL. 1998]: 304-338.
- WALTON ET AL. 1987 Walton, J.D.; Musen, M.A.; Combs, D.M.; Lane, C.D.; Shortliffe, E.H.; Fagan, L.M. (1987): *Graphical Access to Medical Expert Systems III: Design of a Knowledge Acquisition Environment*. Methods of Information in Medicine 26(3): 78-88.
- WANG ET AL. 1997 Wang, X.S.; Bettini, C.; Brodsky, A.; Jajodia, S. (1997): *Logical Design for Temporal Databases with Multiple Granularities*. ACM Transactions on Database Systems 22(2):115-170.
- WESKE 1999 A Weske, M. (1999): *Workflow Management through Distributed and Persistent CORBA Workflow Objects*. In: [JARKE & OBERWEIS 1999]: 446-450.
- WESKE 1999 B Weske, M. (1999): *State-based Modeling of Flexible Workflow Executions in Distributed Environments*. Journal of Integrated Design and Process Science 3(2): 49-62.
- WESKE 2000 C Weske, M. (2000): *Workflow Management Systems: Formal Foundation, Conceptual Design, Implementation Aspects*. Habilitationsschrift Fachbereich Mathematik und Informatik, University of Münster, Germany.
- WESKE ET AL. 1998 Weske, M.; Hündling, J.; Kuropka, D.; Schuschel, H. (1998): *Objektorientierter Entwurf eines flexiblen Workflow-Management-Systems*. Informatik Forschung und Entwicklung 13(4): 179-195.
- WFMC 2002 Workflow Management Coalition (2002): *The Workflow Reference Model*. <http://www.wfmc.org/standards/standards.htm>.
- WIDOM & CERI 1996 Widom, J.; Ceri, S. (Eds.) (1996): *Active Database Systems: Triggers and Rules For Advanced Database Processing*. Morgan Kaufmann, San Francisco, CA, USA.
- WIEDEMANN ET AL. 1998 Wiedemann, T.; Knaup, P.; Bachert, A.; Creutzig, U.; Haux, R.; Schilling, F. (1998): *Computer-aided Documentation and Therapy Planning in Pediatric Oncology*. Proceedings 9th World Congress on Medical Informatics (MEDINFO'98), Seoul, South-Korea: 1306-1309.

-
- WIEST & LEVY 1977 Wiest, J.D.; Levy, F.K. (1997): *A Management Guide to PERT/CPM*. Prentice Hall, Upper Saddle River, NJ, USA.
- WILKINS 1988 Wilkins D.E. (1988): *Practical Planning: Extending the Classical AI Planning Paradigm*. Morgan Kaufmann, San Fransisco, CA, USA.
- WILKINS ET AL. 1995 Wilkins, D. E.; Myers, K. L.; Lowrance, J. D.; Wesley, L. P. (1995): *Planning and Reacting in Uncertain and Dynamic Environments*. Journal of Experimental and Theoretical AI 7(1): 197-227.
- WODTKE & WEIKUM 1997 Wodtke, D.; Weikum, G. (1997): *A Formal Foundation for Distributed Workflow Execution Based on State Charts*. 6th International Conference on Database Theory (ICDT '97), Delphi, Greece (LNCS 1186). Springer, Berlin, Germany: 230-246.
- WOLFF 1999 Wolff, Ch. (1999): *Einführung in Java. Objektorientiertes Programmieren mit der Java 2-Plattform*. Teubner, Stuttgart, Germany.
- WOOLDRIDGE 1997 Wooldridge, M. (1997). *Agent-based Software Engineering*. IEE Proceedings on Software Engineering 144(1): 26-37.
- WOOLDRIDGE & JENNINGS 1995 A Wooldridge, M.; Jennings, N.R. (1995): *Agent Theories, Architectures, and Languages: a Survey*. In: [WOOLDRIDGE & JENNINGS 1995 B]: 1-22.
- WOOLDRIDGE & JENNINGS 1995 B Wooldridge, M.J.; Jennings, N.R. (Eds.) (1995): *Intelligent Agents*. Proceedings Workshop on Agent Theories, Architectures, and Languages (ATAL'94), Amsterdam, The Netherlands (LNCS 890). Springer, Berlin, Germany.
- WORAH & SHETH 1997 Worah, D.; Sheth, A. (1997): *Transactions in Transactional Workflows*. In: [JAJODIA & KERSCHBERG 1997]: 3-34.

[X,Y,Z]

- X/OPEN 1996 X/Open Consortium (1996): *X/Open Distributed Transaction Processing: Reference Model Version 3*. X/Open Company Ltd X/Open Company Ltd, Reading, UK.
- YANG 1997 Yang, Q. (1997): *Intelligent Planning: A Decomposition and Abstraction Based Approach*. Springer, Berlin, Germany.
- YANG & KIFER 2001 Yang, G; Kifer, M. (2001): *FLORA-2 : User's Manual*. Department of Computer Science State University of New York, NY, USA. (<http://xsb.sourceforge.net/>).
- ZHOU ET AL. 1999 Zhou, T.; Liu, L.; Pu, C. (1999): *TAM: A System for Dynamic Transactional Activity Management*. Proceedings ACM International Conference on Management of Data (SIGMOD'99), Philadelphia, PA, USA: 571-573.