Architecture Description for Mobile Distributed Systems

Volker Gruhn and Clemens Schäfer

University of Leipzig
Faculty of Mathematics and Computer Science
Chair for Applied Telematics / e-Business*
Klostergasse 3, 04109 Leipzig, Germany
{gruhn,schaefer}@ebus.informatik.uni-leipzig.de

Abstract. In this paper we motivate an Architecture Description Language (ADL) for mobile distributed systems based on the π-calculus. Different from other approaches, the non-functional properties, which are essential when mobile architectures are described, are treated in a flexible manner by inserting logical formulae for expressing and checking non-functional properties into π-calculus processes. A formal example is given to illustrate the approach before the constituents of the ADL are sketched.

1 Motivation

Modeling the architecture of mobile distributed systems using a domain-specific architecture description language (ADL) is considered as an useful approach [1], since the influence of mobility emphasizes the necessity to examine functional properties of software architectures as well as non-functional properties. This corresponds to the fact that "mobility represents a total meltdown of all stability assumptions ... associated with distributed computing" [2], which subsumes the problems software engineers have to face in practice when they build mobile distributed systems. Examples for these problems are network structures, which are no longer fixed and where nodes may come and go, communication failures due to lost links over wireless networks, or restricted connectivity due to low bandwidth of mobile communications links. These all have in common that they affect the non-functional properties of a system like performance, robustness, security, or quality of service. Besides non-functional properties, these intrinsic challenges of mobile systems may also affect the functional aspects of a system, since a mobile system may have to provide extra functionality (like replication facilities, caching mechanisms etc.) in order to ensure usability in situations where the aforementioned problems occur. With *Con Moto* (Italian for "in motion") we propose an ADL which enables system developers to address these issues during the early stages of system development in order to allow them to make appropriate design choices for mobile systems.

* The chair for Applied Telematics/e-Business is endowed by Deutsche Telekom AG.

2 Introduction

ADLs have been area of research for many years. It is commonly understood that an ADL comprises three essential constituents: components, connectors and configurations [3]. Roughly speaking, components model the entities of software systems which perform computations or store data, connectors model the interaction of components, and configurations are connected graphs of components and connectors. Based on this understanding and the motivation given before, we can list the requirements for an ADL for mobile distributed systems:

- A mobile ADL must be able to model dynamic aspects of a system like the dynamic instantiation of components or the change of communication links during system execution.
- A mobile ADL should be able to model different communication channels with non-functional properties like reliability or bandwidth. This is necessary to analyze systems and to find possible problems that might arise when a connection fails. Therefore specialized connectors might be necessary.
- A mobile ADL should allow the composition of non-functional properties in order to be able to model the complex dependencies which are prominent in mobile distributed systems.
- A mobile ADL should be formally based, so that simulation and reasoning about the model is possible.

With Con Moto we strive to fulfill these requirements. The remainder of this paper is structured as follows. After an overview of the related work in section 3, an introduction in π-calculus (section 4.1) is given which acts as basis for the formal example in section 4.2, which illustrates the core concept of our considerations. After depicting the use of the formal model in Con Moto (section 5), a conclusion is drawn.

3 Related Work

ADLs in general have been topic of research in previous years. The necessity for modeling non-functional properties in architecture description has been recognized by Shaw and Garlan [4]. The classification work of Medvidovic and Taylor [5] present a sound compilation of properties of ADLs. From their work it becomes obvious, that none of the ADLs presented there is suitable for modeling dynamic aspects of mobile systems. In the past, this fact lead to the development of mobile ADLs which have recently been presented. The ArchWare project with its π-ADL [6] is one result of these efforts. Another mobile ADL can be found in the works of Issarny et al. [7]. Both present an ADL for mobile systems based on Milner's π-calculus [8]. These ADLs have in common that they are able model the dynamics of mobile systems, which is due to their theoretical foundation in the π-calculus. Although they vary in terms of elaboration and tool support, the fundamental difference—from the perspective of this paper—is the treatment of non-functional properties, which is absent in the π-ADL approach.

Issarny addresses non-functional properties in her work, but the treatment of non-functional properties is bound to a global conformance condition, which must hold for a predefined set of non-functional properties assigned to components and connectors, and does not allow the composition of non-functional properties, which is novel in our approach. Currently, there is other research in the area non-functional properties of software systems. This work is mainly based on the Lamport's TLA+ language [9], which is a logic for specifying and reasoning about concurrent and reactive systems. Zschaler [10] presents a specification of timeliness properties of component based systems, but these as well as the underlying work of Aagedal [11], where the integration of TLA+ approach into architectural description is proposed, the models in TLA+ lack the support for mobility and are thus not regarded further.

4 System Model

4.1 Use of π-Calculus

Similar to the approach of Issarny et al. [7], we base Con Moto on a service-oriented interaction paradigm, i.e. a component abstracts a networked service which invokes operations of peer components and dually executes operations that are invoked. Processes are the foundation for grasping the functional aspects of the architectural description. Since we use Milner's π-calculus [8] for modeling, we give a very brief introduction into the monadic π-calculus (c.f. [12]) first: The simplest entities of the π-calculus are *names*. These can be seen as names of communication links and used by processes for interaction. These processes evolve by performing actions. Capability for actions are expressed as *prefixes*, of which we use three kinds[1]:

$$\pi ::= \overline{x}(y) \;\mid\; x(z) \;\mid\; [x = y]\pi \ . \tag{1}$$

The first capability is to send the name y via the name x, and the second to receive a name via x. The third is a conditional capability: the capability π if x and y are the same name. The *processes* and *summations* of the π-calculus are given by:

$$P ::= M \;\mid\; P \,|\, P' \;\mid\; !P \tag{2}$$

$$M ::= \mathbf{0} \;\mid\; \pi.P \;\mid\; M + M' \;\mid\; \mathbf{1}. \tag{3}$$

The semantics are as follows. $\mathbf{0}$ means inaction, the prefix $\pi.P$ means that P can be executed after π has been exercised; the sum $M + M'$ models a choice, the composition $P|P'$ is known as parallelism; $!P$ means replication. $\mathbf{1}$ is an extension by ourselves and has the notion of a "dummy" process: A process that can always be executed and does not perform any actions. We need this extension in our

[1] We omit the non-observable action τ and binding of names for shortness.

later example.[2] However, for modeling non-functional properties it is not enough to just exchange names between processes. We therefore make use of the polyadic π-calculus, which extends the monadic π-calculus in that way that tuples can be passed by actions instead of names. This leads to the following prefixes

$$\pi ::= \overline{x}(\widetilde{y}) \ \mid \ x(\widetilde{z}) \ \mid \ [x = y]\pi, \tag{4}$$

where no names occur more than once in the tuple $\widetilde{z}$ in an input prefix. In the following example we will use this polyadic π-calculus to illustrate our core idea. However, an formally exact treatment of this issue would require the usage of typed π-calculi, which we omit here for the sake of readability.

4.2 Formal Example

As in Issarny's work [7], we use processes given in π-calculus for expressing the functional properties of our architecture. We now extend the processes to cover also non-functional properties. The core idea behind this approach is, that every action in our processes can return its non-functional properties like execution time, memory consumption, availability etc. We will now introduce two components and their services and will show how their non-functional properties can be handled. However, we show the treatment only for abstract non-functional properties, since concrete properties would increase formal complexity, but would not contribute to the core idea.

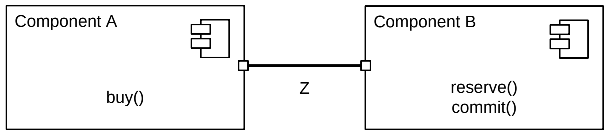

Fig. 1. Example components in UML-like notation

Assume the following scenario: as intuitively depicted in Figure 1 we have two components A and B. A offers the service *buy*(), whereas B offers the services *reserve*() and *commit*(), which are subsequently invoked during the execution of *buy*(). Since *reserve*() and *commit*() have a certain set of non-functional properties, it is intuitively clear that the non-functional properties of *buy*() should be a composition of the properties of *reserve*() and *commit*(). If we leave away all other aspects and just model the functional behaviour of A and B, we write in monadic π-calculus:

$$P_B \overset{\text{def}}{=} reserve(x).\overline{reserve}(x).\mathbf{0} \ \mid \ commit(x).\overline{commit}(x).\mathbf{0} \tag{5}$$

$$P_A \overset{\text{def}}{=} buy(x).\overline{reserve}(x).reserve(x).\overline{commit}(x).commit(x).\overline{buy}(x).\mathbf{0} \tag{6}$$

[2] Although $\mathbf{1}$ is formally not absolutely necessary for our modeling purposes, it enhances readability in the later examples. Formally we define the following reaction for our "dummy" process: $\mathbf{1}.\pi \rightarrow \pi$.

```
component A: { provides { buy()     nfprop α() }
               requires { reserve() ensure β′()
                          commit()  ensure γ′() } }
component B: { provides { reserve() nfprop β()
                          commit()  nfprop γ() }
               requires { ∅ } }
connector Z: {                      nfprop ζ() }
```

Fig. 2. Example components in textual notation

The process P_B models the behavior of component B and the process P_A for the component A. For invocation of the service *buy*() (which we assume is modeled by reading a value by $buy(x)$), an output $\overline{reserve}(x)$ is made to the processes in component B which models the invocation of *reserve*(). After *reserve*() has returned (the input operation $reserve(x)$), $\overline{commit}()$ is invoked similarly. Finally, *buy*() returns. This is modeled by the output $\overline{buy}(x)$.

We now introduce the non-functional properties. The idea is as follows: Every service returns its non-functional properties when it terminates. In the textual notation in Figure 2, the keyword `nfprop` indicates a function which computes the non-functional properties of a given service (e.g. $\alpha()$ evaluates to the non-functional properties of *buy*()). These functions are defined for all services a component provides, which are listed after the keyword `provides`. Since non-functional properties have to be checked throughout the execution of the system (which refers to the global conformance condition in the work of Issarny), we also introduce a function for each service required by a component (indicated by the keyword `requires` in the example), which grasps the non-functional requirements for the service and therefore evaluates to true if these requirements are met. These functions are also given in the example after the keyword `ensure`. In our example, $\beta'()$ models the non-functional requirements for *reserve*() in component A. For completeness, we now also model the connector Z, through which the services of B are invoked. This connector also has a function $\zeta()$ to determine its non-functional properties. We now integrate the functions for computing and checking non-functional properties into our examples 5 and 6:

$$P'_B \overset{\text{def}}{=} reserve(x).\overline{reserve}(\langle x, \beta()\rangle).\mathbf{0} \;\mid\; commit(x).\overline{commit}(\langle x, \gamma()\rangle).\mathbf{0} \quad (7)$$

$$P'_A \overset{\text{def}}{=} buy(x).\overline{reserve}(x).reserve(\langle x, p\rangle).[\beta'(p)]\mathbf{1}.$$
$$\overline{commit}(x).commit(\langle x, q\rangle).[\gamma'(q)]\mathbf{1}.\overline{buy}(\langle x, \alpha(p, q)\rangle).\mathbf{0} \quad (8)$$

Now, *reserve*() is invoked as earlier. However, *reserve*() returns a tuple, the name x as before and its non-functional properties p. Now, in the execution of *buy*() it is checked, whether the requirement β' holds for the properties p. If this is the case, the process can continue by executing the "dummy"-process **1**. The same two steps are performed for *commit*(). Finally, the function α is evaluated in order to retrieve the composed non-functional property of *buy*() and returned in the extended output statement. If we want to model the influence of the connector Z, we have to use its transfer function $\zeta()$ and apply it to the non-functional properties returned by *reserve*() and *commit*(), i.e. we have to replace all occurrences of p and q with $\zeta(p)$ and $\zeta(q)$ respectively. Therefore, our process from 8 is transformed into

$$\begin{aligned} P''_A \overset{\text{def}}{=} & \, buy(x).\overline{reserve}(x).reserve(\langle x, p\rangle).[\beta'(\zeta(p))]\mathbf{1}. \\ & \overline{commit}(x).commit(\langle x, q\rangle).[\gamma'(\zeta(q))]\mathbf{1}.\overline{buy}(\langle x, \alpha(\zeta(p), \zeta(q))\rangle).\mathbf{0} \end{aligned} \quad (9)$$

Comparing the formulae 6 and 9, we see that the pure functional modeling of the behavior of component A could be evolved to a specification which includes abstract non-functional properties, allowing their composition and checking. This was achieved by subsequently applying transformation steps and enriching the formal functional specification.

5 Use of Model in Con Moto

In the following section we will discuss how the presented approach for modeling non-functional properties will be used in the ADL Con Moto. Here, models of software systems need to be given in a textual representation as indicated in Figure 2. However, in order to ease system composition, Con Moto will also provide a graphical representation which is based on concepts of UML 2.0 for modeling software architecture, which allows the use of components, ports and connectors. An example of a architectural diagram in UML style is given in the Figure 1.

In the textual representation, there is also the need for expressing the functional properties of the system, hence the invocations of processes, which can be compiled into π-calculus processes like those we used in the example. This is work which has to be done by the system designers, since the functional aspects are crucial for the modeling of mobile systems. Additionally, the designers have to provide the functions evaluating and checking the non-functional properties.

The composition of the processes as in our example can be done automatically by the Con Moto environment, so that for the designer there is the clear separation between functional and non-functional aspects in order to keep modeling complexity at a low level. After the Con Moto environment has composed the functional and non-functional properties into a enriched π-calculus specification, there is the model which allows checking.

A general useful approach for checking π-calculus models for certain properties is to apply model checking techniques. There are rather straight-forward transformations which allow the generation of input for model checkers from π-calculus models. One transformation of this kind is presented in the work of Song and Compton [13]. They propose a formalism for converting π-calculus models into the Promela language used by the SPIN model checker [14]. Although in their paper, Song and Compton restrict their transformation to monadic π-calculus, an extension to polyadic and typed π-calculus is possible. Our approach of integrating conditions for non-functional properties can also be added to the approach presented in [13]. Although it should be noted, that mapping the free conditions to Promela makes restrictions of this language apply to our conditions. But we are confident, that the power of Promela is sufficient for our modeling purposes.

It should be emphasized that we did not make any conclusions about complexity of a Con Moto model with regard to model checking yet. It can easily be imagined that choosing certain non-functional property definitions can lead to a state explosion in the model checker which makes checking of the model impossible. Nevertheless, since a Promela representation of the model also allows the simulation of the model, certain aspects of the architecture can also be checked by simulation.

6 Conclusion

We presented a formal foundation for modeling non-functional properties in architectural description. The main contribution to the research is that it facilitates a general treatment of non-functional properties, ensuring compositionality aspects and flexible checking, which provides a powerful tool for specifying mobile dynamic systems. After motivating our approach we showed that it is possible to pass non-functional properties in π-calculus processes. Since we enriched these processes with checking conditions, it is possible to extend the existing approaches for mobile ADLs with a general treatment of non-functional properties and hence prepare the groundwork for our ADL Con Moto.

Ongoing work is to elaborate the formal underpinning of the chosen approach: The approach has to be written down in a formal correct way using polyadic typed π-calculus, and properties of the extended notion of π-calculus processes have to be proven. The mapping of π-calculus to Promela has to be finished in order to provide tool support. Furthermore, an Eclipse plugin is in work which will allow the integration of architecture modeling with Con Moto into the accepted development process. Summing up, we are confident, that these contributions can add substantial benefit to the early stages of mobile system design.

References

1. Gruhn, V., Schäfer, C.: An Architecture Description Language for Mobile Distributed Systems. In: Proceedings of the First European Workshop on Software Architecture (EWSA 2004), Springer-Verlag Berlin Heidelberg (2004) 212–218
2. Roman, G.C., Picco, G.P., Murphy, A.L.: Software Engineering for Mobility: A Roadmap. In: Proceedings of the Conference on the Future of Software Engineering, ACM Press (2000) 241–258
3. Medvidovic, N., Rosenblum, D.S.: Domains of Concern in Software Architectures. In: Proceedings of the 1997 USENIX Conference on Domain-Specific Languages. (1997)
4. Shaw, M., Garlan, D.: Formulations and Formalisms in Software Architecture. In van Leeuwen, J., ed.: Computer Science Today: Recent Trends and Developments. Volume 1000 of Lecture Notes in Computer Science., Springer (1995) 307–323

5. Medvidovic, N., Taylor, R.N.: A Classification and Comparison Framework for Software Architecture Description Languages. IEEE Transactions on Software Engineering **26** (2000) 70–93
6. Oquendo, F.: π-ADL: An Architecture Description Language based on the Higher-Order Typed π-Calculus for Specifying Dynamic and Mobile Software Architectures. ACM Software Engineering Notes **29** (2004)
7. Issarny, V., Tartanoglu, F., Liu, J., Sailhan, F.: Software Architecture for Mobile Distributed Computing. In: Proceedings of the Fourth Working IEEE/IFIP Conference on Software Architecture (WICSA'04), IEEE (2004) 201–210
8. Milner, R.: Communicating and Mobile Systems: the π-Calculus. Cambridge University Press (1999)
9. Lamport, L.: Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers. Addison-Wesley (2002)
10. Zschaler, S.: Formal specification of non-functional properties of component-based software. In Bruel, J.M., Georg, G., Hussmann, H., Ober, I., Pohl, C., Whittle, J., Zschaler, S., eds.: Workshop on Models for Non-functional Aspects of Component-Based Software (NfC'04) at UML conference 2004. (2004)
11. Aagedal, J.Ø.: Quality of Service Support in Development of Distributed Systems. PhD thesis, University of Oslo (2001)
12. Sangiorgi, D., Walker, D.: The π-calculus: a Theory of Mobile Processes. Cambridge University Press (2001)
13. Song, H., Compton, K.J.: Verifying π-calculus Processes by Promela Translation. Technical Report CSE-TR-472-03, University of Michigan (2003)
14. Holzmann, G.J.: The Spin Model Checker: Primer and Reference Manual. Addison-Wesley (2004)