Architecture Description for Mobile Distributed Systems Using Typed π-Calculus

Volker Gruhn¹, Clemens Schäfer²

Chair for Applied Telematics / e-Business³
Faculty of Mathematics and Computer Science
University of Leipzig, Germany

Abstract
In this paper we motivate an Architecture Description Language (ADL) for mobile distributed systems based on typed 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 typed π-calculus processes. A formal example is given to illustrate the approach before the constituents of the ADL are sketched.

Keywords: Specifications, Requirements, Model checking, Mobile Systems, ADL

1 Motivation
Modeling the architecture of mobile distributed systems using a domain-specific architecture description language (ADL) is considered as an useful approach [2], 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” [10], which subsumes the problems software engineers have to face in practice when they

¹ Email: gruhn@ebus.informatik.uni-leipzig.de
² Email: schaefer@ebus.informatik.uni-leipzig.de
³ The chair for Applied Telematics/e-Business is endowed by Deutsche Telekom AG.

1571-0661 © 2006 Elsevier B.V. Open access under CC BY-NC-ND license.
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 our ADL Con Moto (Italian for “in motion”) we propose a language 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 the mobile system.

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 [6]. Roughly speaking, components model the entities of software systems which perform computations or which 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 overview of the related work in section
3, an introduction in \(\pi\)-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 a topic of research in previous years. The necessity for modeling non-functional properties in architecture description has been recognized by Shaw and Garlan [12]. The classification work of Medvidovic and Taylor [7] presents a sound compilation of properties of existing 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 \(\pi\)-ADL [9] is one result of these efforts. Another mobile ADL can be found in the works of Issarny et al. [4]. Both present an ADL for mobile systems based on Milner’s \(\pi\)-calculus [8]. These two ADLs have in common that they are able to model the dynamics of mobile systems, which is due to their theoretical foundation in the \(\pi\)-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 \(\pi\)-ADL approach. Issarny et al. address non-functional properties in their 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 [5], which is a logic for specifying and reasoning about concurrent and reactive systems. Zschaler [14] presents a specification of timeliness properties of component based systems, but these as well as the underlying work of Aagedal [1], where the integration of TLA+ approach into architectural description is proposed, are not further regarded in our context, since the models in TLA+ lack the support for mobility.

4 System Model

4.1 Use of \(\pi\)-Calculus

Similar to the approach of Issarny et al. [4], 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 $\pi$-calculus [8] for modeling, we give a very brief introduction into the monadic $\pi$-calculus (c.f. [11]) first: The simplest entities of the $\pi$-calculus are names. These can be seen as name of communication links and are used by processes for interaction. These processes evolve by performing actions. Capability for actions are expressed as prefixes, of which we use three kinds$^4$:

\[ \pi ::= x(y) \mid x(z) \mid [x = y]\pi. \]

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 $\pi$ if $x$ and $y$ are the same name. The processes and summations of the $\pi$-calculus are given by:

\[ P ::= M \mid P \mid P' \mid !P \]

\[ M ::= 0 \mid \pi.P \mid M + M' \mid 1. \]

The semantics are as follows. $0$ means inaction, the prefix $\pi.P$ means that $P$ can be executed after $\pi$ has been exercised; the sum $M + M'$ models a choice, the composition $P|P'$ is known as parallelism; $!P$ means replication. $1$ 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 later example.\(^5\) However, for modeling non-functional properties it is not enough to just exchange names between processes. We therefore make use of the polyadic $\pi$-calculus, which extends the monadic $\pi$-calculus in that way that tuples can be passed by actions instead of names.

This leads to the following prefixes

\[ \pi ::= x(y) \mid x(z) \mid [x = y]\pi, \]

where no names occur more than once in the tuple $\tilde{z}$ in an input prefix. In the following example we will use this polyadic $\pi$-calculus together with types to illustrate our core idea.

Later we will use records in $\pi$-calculus notation to model non-functional properties. According [11] we use the following definition of records. Given a set of types $T_1, \ldots, T_n$. Then, a record type has the form

\[ \{\ell_1 : T_1, \ldots, \ell_n : T_n\}, \]

\(^4\) We omit the non-observable action $\tau$ and binding of names for shortness.

\(^5\) Although $1$ 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: $1.\pi \rightarrow \pi$. 
where all labels $\ell_i$ are different. The values are of the form
\[
\{\ell_1 = v_1, \ldots, \ell_n = v_n\},
\]
where each $v_i$ is value of field $\ell_i$. We can model non-functional properties now like:
\[
T_{\text{serviceA}} \overset{\text{def}}{=} \{\text{availability} : \text{real}, \text{resourceConsumption} : \text{real}\}.
\]
In this example, a type for a non-functional property with name $T_{\text{serviceA}}$ is defined which comprises two elements, availability and resourceConsumption; both are considered to be real numbers.

4.2 Formal Example

As in the work of Issarny et al. [4], we use processes given in $\pi$-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.

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

\[
P_B \overset{\text{def}}{=} \text{reserve}(x).\overline{\text{reserve}(x)}.0 \mid \text{commit}(x).\overline{\text{commit}(x)}.0
\]
\[
P_A \overset{\text{def}}{=} \text{buy}(x).\overline{\text{reserve}(x)}.\overline{\text{reserve}(x)}.\text{commit}(x).\overline{\text{commit}(x)}.\overline{\text{buy}(x)}.0
\]
The process $P_B$ models the behavior of component $B$ and the process $P_A$ for the component $A$. For invocation of the service $\text{buy}()$ (which we assume is modeled by reading a value by $\text{buy}(x)$), an output $\overline{\text{reserve}(x)}$ is made to the
component A: { provides { buy() } nfprop α : Tα,in → Tα,out }  
requires { reserve() ensure β′ : Tβ′ → bool  
commit() ensure γ′ : Tγ′ → bool } }  
component B: { provides { reserve() } nfprop β : Tβ,in → Tβ,out  
commit() nfprop γ : Tγ,in → Tγ,out }  
requires { { } }  
connector Z: { } nfprop ζ : Tζ,in → Tζ,out }  

Fig. 2. Example Components in Textual Notation

processes in component B which models the invocation of reserve(). After reserve() has returned (the input operation reserve(x)), commit() is invoked similarly. Finally, buy() returns. This is modeled by the output 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. α() 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 et al.), 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, β′() 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 ζ() to determine its non-functional properties.

For the functions modeling non-functional properties given by the keyword nfprop the input and output types are given in the signature function name : argument type → result type. For the functions checking non-functional properties (keyword ensure) the result type bool with bool = {true, false} is used. The boolean values of true and false are used for the conditional expression in π-calculus processes.

We now integrate the functions for computing and checking non-functional properties into our examples 8 and 9:
\begin{align}
(10) & \quad P_B' \overset{\text{def}}{=} \text{reserve}(x).\text{reserve}(\langle x, \beta() \rangle).0 \mid \text{commit}(x).\text{commit}(\langle x, \gamma() \rangle).0 \\
& \quad P_A' \overset{\text{def}}{=} \text{buy}(x). \\
& \quad \text{reserve}(x).\text{reserve}(\langle x, p : T_{\beta,\text{out}} \rangle).[\beta'(p : T_{\beta'})]1. \\
& \quad \text{commit}(x).\text{commit}(\langle x, q : T_{\gamma,\text{out}} \rangle).[\gamma'(q : T_{\gamma'})]1. \\
& \quad \text{buy}(\langle x, \alpha(\langle p, q \rangle : T_{\alpha,\text{in}}) \rangle)).0 \end{align}

Now, \text{reserve()} is invoked as earlier. However, \text{reserve()} returns a tuple, the name \(x\) as before and its non-functional properties \(p\). Now, in the execution of \text{buy()} it is checked, whether the requirement \(\beta'\) 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 \text{commit()}. Finally, the function \(\alpha\) is evaluated in order to retrieve the composed non-functional property of \text{buy()} and returned in the extended output statement.

In Figure 2 we associated an input type and an output type with all functions computing non-functional properties. To illustrate which types are used in the formula, the notation \(v : T\), where \(v\) is a name, tuple or record and \(T\) is a type, is used. However, due to the composition in formula 11 the following type equivalences must hold in order to allow the composition: \(T_{\beta,\text{out}} = T_{\beta'}\), \(T_{\gamma,\text{out}} = T_{\gamma'}\) and \(\langle T_{\beta,\text{out}}, T_{\gamma,\text{out}} \rangle = T_{\alpha,\text{in}}\).

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 \text{reserve()} and \text{commit()}, i.e. we have to replace all occurrences of \(p\) and \(q\) with \(\zeta(p)\) and \(\zeta(q)\) respectively. Therefore, our process from 11 is transformed into

\begin{align}
(12) & \quad P_A'' \overset{\text{def}}{=} \text{buy}(x). \\
& \quad \text{reserve}(x).\text{reserve}(\langle x, p : T_{\beta,\text{out}} \rangle).[\beta'(\zeta(p : T_{\zeta,\text{in}})))]1. \\
& \quad \text{commit}(x).\text{commit}(\langle x, q : T_{\gamma,\text{out}} \rangle).[\gamma'(\zeta(q : T_{\zeta,\text{in}})))]1. \\
& \quad \text{buy}(\langle x, \alpha(\langle \zeta(p : T_{\zeta,\text{in}}), \zeta(q : T_{\zeta,\text{in}}) \rangle) \rangle)).0 \\
\end{align}

In addition to the type constraints which applied for formula 11, here in formula 12 the following type constraints have to be obeyed in order to allow the composition: \(T_{\beta,\text{out}} = T_{\zeta,\text{in}}, T_{\gamma,\text{out}} = T_{\zeta,\text{in}}, T_{\zeta,\text{out}} = T_{\beta'} = T_{\gamma'}\) and \(T_{\alpha,\text{in}} = \langle T_{\zeta,\text{out}}, T_{\zeta,\text{out}} \rangle\).

Comparing the formulae 9 and 12, 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 to $\pi$-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 an enriched $\pi$-calculus specification, there is the model which allows checking.

A general useful approach for checking $\pi$-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 $\pi$-calculus models. One transformation of this kind is presented in the work of Song and Compton [13]. They propose a formalism for converting $\pi$-calculus models into the Promela language used by the SPIN model checker [3]. Although in their paper, Song and Compton restrict their transformation to monadic $\pi$-calculus, an extension to polyadic and typed $\pi$-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 marked, 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 typed $\pi$-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, which is currently in progress, is to elaborate the formal underpinning of the chosen approach: The approach has to be written down in a sound formal way and properties of the extended notion of $\pi$-calculus processes have to be proven. The mapping of $\pi$-calculus to Promela, which is worked out at the moment, 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


