Specifying Properties of Dynamic Architectures Using Configuration Traces

Diego Marmsoler(✉) and Mario Gleirscher

Technische Universität München, Munich, Germany
{diego.marmsoler,mario.gleirscher}@tum.de

Abstract. The architecture of a system describes the system's overall organization into components and connections between those components. With the emergence of mobile computing, dynamic architectures became increasingly important. In such architectures, components may appear or disappear, and connections may change over time.

Despite the growing importance of dynamic architectures, the specification of properties for those architectures remains a challenge. To address this problem, we introduce the notion of *configuration traces* to model properties of dynamic architectures. Then, we investigate these properties to identify different types thereof. We show *completeness* and *consistency* of these types, i.e., we show that (almost) every property can be separated into these types and that a property of one type does not impact properties of other types.

Configuration traces can be used to specify general properties of dynamic architectures and the separation into different types provides a systematic way for their specification. To evaluate our approach we apply it to the specification and verification of the Blackboard pattern in Isabelle/HOL.

1 Introduction

A systems architecture provides a set of components and connections between their ports. With the emergence of mobile computing, dynamic architectures became more and more important [5,10,20]. In such architectures, components can appear and disappear and connections can change, both over time.

Despite the increasing importance of dynamic architectures some questions regarding their specification still remain:

– How can *properties* of dynamic architectures be specified in general?
– How can those properties be separated into different *types*?

A property of dynamic architectures characterizes executions of such architectures. Consider, for example, the following property for a publisher-subscriber [8] system: *Whenever a component p of type* `Publisher` *provides a message for which a* `Subscriber` *component s was subscribed, s is connected to p.* Another example describes a property of a Blackboard architecture [8]: *Whenever a component of type* `BlackBoard` *provides a message containing a problem to be solved,*

A. Sampaio and F. Wang (Eds.): ICTAC 2016, LNCS 9965, pp. 235–254, 2016.
DOI: 10.1007/978-3-319-46750-4_14

a component of type `KnowledgeSource`*, able to solve this problem, is eventually activated.* Usually, such properties can be separated into different types, such as: (i) *Behavior properties* characterizing the behavior of certain components. (ii) *Activation properties* characterizing the activation/deactivation of components. (iii) *Connection properties* characterizing the dynamic connection between components.

To answer the above questions, we first introduce a formal model of dynamic architectures. Thereby we model an architecture as a set of configuration traces which, in turn, is a sequence over architecture configurations. An architecture configuration is modeled as a set of components, valuations of the component ports with messages, and connections between these ports.

In a second step, we characterize *behavior*, *activation*, and *connection* properties. We show the distinct nature of those types of properties and investigate their expressive power. Thereby we characterize the notion of *separable architecture property* and show that each of them can be *uniquely* described through the intersection of a corresponding behavior, activation, and connection property.

We evaluate our approach by specifying (and analyzing) the Blackboard pattern for dynamic architectures using the Isabelle/HOL [22] interactive theorem prover. Therefore, we first specify behavior, activation, and connection properties for Blackboard architectures. Then, we specify the pattern's guarantee as an architecture property. Finally, we verify the pattern by proving its guarantee from the original properties using Isabelle's structured proof language Isar [28].

The remainder of the article is organized as follows: Sect. 2 reviews existing work in this area. Section 3 introduces the Blackboard pattern as a running example. Section 4 introduces our model for dynamic architectures. Section 5 describes and investigates different types of properties for those architectures. Section 6 presents an approach to systematically specify properties and applies it to specify the Blackboard pattern. In Sect. 7 we provide a critical discussion of possible weaknesses of the approach. Finally, Sect. 8 summarizes our results and discusses potential implications and future work.

2 Background and Related Work

Related work can be found in three different areas: 1. Architecture Description Languages, 2. Modeling of Architectural Styles, and 3. Modeling of Constraints for Dynamic Architectures. In the following we briefly discuss each of them.

2.1 Architecture Description Languages

Over the last three decades, a number of so-called Architecture Description Languages (ADLs) emerged to support the formal specification of architectures. Some of them also support the specification of dynamic aspects such as Rapide [17], Darwin [18], Dynamic Wright [2,3], Π-ADL [23], xADL [11], and ACME [13].

While ADLs support the formal specification of architectures, they were developed with the aim to specify individual architecture solutions, rather than properties for architectures which require more abstract specification techniques. Nevertheless, these works provide the conceptual foundation for our work since many of the abstractions used in our model are based on the concepts introduced by ADLs.

2.2 Modeling Architectural Styles

Architectural styles focus on the specification of architectural constraints, rather than specific architectures.

One of the first approaches to formalize architectural styles is discussed by Abowd et al. [1]. There, the authors apply a denotational semantics approach to software architectures by using the specification language Z [26]. Other examples used to specify architectural styles include the Chemical Abstract Machine [15] or Wright [3] which allow for the specification of architectural constraints for static architectures. Two further ideas come from Moriconi et al. [21] and Penix et al. [24]. Both apply algebraic specification to software architectures. Finally, Bernardo et al. [4] use process algebras to specify architectural types which can be seen as a form of architectural styles.

While these approaches focus on the specification of architectural constraints rather than architectures, they do usually not allow for the specification of dynamic architectural constraints which is the focus of this work. Nevertheless, these works provide many important conceptual insights into the specification of architectural constraints on which we build.

2.3 Specification of Constraints for Dynamic Architectures

Work in this area is most closely related to our work.

The approach of Le Métayer [16] applies graph theory to specify architectural evolution. The author proposes the use of graph grammars to specify architectural evolution. A similar approach comes from Hirsch and Montanari [14] who employ hypergraphs as a formal model to represent styles and their reconfigurations. While we also apply a graph-based approach to model architectural properties, the major difference lies in the specification of behavior. While the discussed approaches focus on structural aspects, we aim at a combination of structural and behavioral aspects.

Another, closely related approach is the one of Wermlinger et al. [29]. The authors combine behavior and structure to model dynamic reconfigurations. One major difference to our work concerns the underlying model of interaction. While the authors use an action synchronization communication model, our model is based on time-synchronous communication. Both communication models have their advantages and drawbacks. Thus, by providing a time-synchronous alternative, we actually complement their work.

Recently, categorical approaches to dynamic architecture reconfiguration appeared such as the work of Castro et al. [9] or Fiadeiro and Lopes [12].

While these approaches provide fundamental insights into the specification of dynamic architecture properties, their model remains implicitly in the categorical constructions. Thus, we complement their work by providing an explicit model of dynamic architecture properties.

Finally, we do not know of any existing work investigating different types of properties of dynamic architectures. However, as stated in the introduction, this is an important aspect to systematically specify properties of dynamic architectures. In this work we provide a formal investigation of properties which is another contribution to current literature.

3 Running Example: Specifying Blackboard Architectures

In this paper, we use the Blackboard architecture design pattern as a running example to show our approach to the specification and verification of dynamic architectures. This pattern was described, for example, by Shaw and Garlan [25], Buschmann et al. [8], and Taylor et al. [27].

Blackboards work with *problems* and *solutions* for them. Hence, we denote by `PROB` the set of all problems and by `SOL` the set of all solutions. Complex problems consist of *subproblems* which can be complex themselves. To solve a problem, its subproblems have to be solved first. Therefore, we assume the existence of a *subproblem relation* $\prec \subseteq \texttt{PROB} \times \texttt{PROB}$. For complex problems, this relation may not be known in advance. Indeed, one of the benefits of a Blackboard architecture is that a problem can be solved also without knowing this relation in advance. However, the subproblem relation has to be well-founded (*wf*) for a problem to be solvable. In particular, we do not allow cycles in the transitive closure of $\prec$.

While there may be different approaches to solve a problem (i.e. several ways to split a problem into subproblems), we assume that the final solution for a problem is unique. Thus, we assume the existence of a function $\texttt{solve}\colon \texttt{PROB} \rightarrow \texttt{SOL}$ which assigns the *correct* solution to each problem. Note, however, that this function is not known in advance and it is one of the reasons of using this pattern to calculate this function.

4 A Model of Dynamic Architectures

In the following we introduce our model of dynamic architectures. It is based on Broy's Focus theory [6] and an adaptation of its dynamic extension [7]. A property is modeled as a set of *configuration traces* which are sequences of *architecture configurations* that, in turn, consist of a set of *active components*, valuation of their ports with type-conform messages, and *connections* between their ports. The model serves the specification of properties for dynamic architectures as shown by the running example.

4.1 Foundations

This section introduces basic concepts of our model such as ports which can be valuated by messages.

Convention 1. *In the following, we denote by* $X \dashrightarrow Y$*, the set of partial functions from a set* X *to a set* Y*. For a partial function* $f\colon X \dashrightarrow Y$*, we denote by:*

- $\operatorname{dom}(f)$ *the domain of* f*,*
- $\operatorname{ran}(f)$ *the range of* f*, and by*
- $f\,|_{X'}$ *the restriction of* f *to the set* $X' \subseteq X$*. If* $X = \mathbb{N}$ *and* $x \in \mathbb{N}$ *we denote by* $f \downarrow_x \stackrel{def}{=} f\,|_{\{n \in \mathbb{N} \mid n \leqslant x\}}$ *the restriction of* f *to the first* x *numbers.*

If $\operatorname{dom}(f) = X$*,* f *is called* total *and denoted by* $f\colon X \to Y$*.*

Messages and ports. In our model, components communicate by exchanging *messages* over *ports*. Thereby, ports are typed by a set of messages which can go through the corresponding port. Thus, we assume the existence of the following sets:

- set M containing all messages,
- sets $\mathsf{P_i}$ and $\mathsf{P_o}$ containing all input and output ports, respectively, and set $\mathsf{P} = \mathsf{P_i} \cup \mathsf{P_o}$ containing all ports. We require a port to be either input or output, but not both:

$$\mathsf{P_i} \cap \mathsf{P_o} = \varnothing. \tag{1}$$

Moreover, we assume the existence of a type function which assigns a set of messages to each port:

$$(T_p)_{p \in \mathsf{P}}, \text{ with } T_p \subseteq \mathsf{M} \text{ for each } p \in \mathsf{P}. \tag{2}$$

Valuation. In our model, components communicate by sending and receiving messages through ports. This is achieved through the notion of *port valuation*. Roughly speaking, a valuation for a set of ports is an assignment of messages to each port. Note that in our model, ports can be valuated by a set of messages meaning that a component can send/receive no message, a single message, or multiple messages at each point in time.

For ports $P \subseteq \mathsf{P}$, we denote by $\overline{P}$ the set of all possible *port-valuations*, formally,

$$\overline{P} \stackrel{\text{def}}{=} \{\mu\colon P \to \wp(\mathsf{M}) \mid \forall p \in P\colon \mu(p) \subseteq T_p\}. \tag{3}$$

Moreover, we denote by $[p_1, p_2, \ldots \mapsto \{m_1\}, \{m_2\}, \ldots]$ the valuation of ports $p_1, p_2, \ldots$ with sets $\{m_1\}, \{m_2\}, \ldots$, respectively. For singleton sets we shall sometimes omit the set parentheses and simply write $[p_1, p_2, \ldots \mapsto m_1, m_2, \ldots]$.

4.2 Components and Interfaces

This section introduces the basic notions of component and interface.

Components. In our model, the basic unit of computation is a *component*. A component is identified by a component identifier which is why we postulate the existence of the set of all component identifiers C.

Component port valuation. In our model, the same port can be reused by different components. Thus, to uniquely identify a *component port*, we need to combine it with the corresponding component. Therefore, we generalize the notion of port valuation introduced in Eq. (3) to component ports $P \subseteq \mathsf{C} \times \mathsf{P}$ as follows:

$$\overline{P} \stackrel{\text{def}}{=} \{\mu \colon P \to \wp(\mathsf{M}) \mid \forall (c,p) \in P \colon \mu((c,p)) \subseteq T_p\}.$$

Interfaces. A component communicates with its environment through an interface by sending and receiving messages over ports.

Definition 2. *An* interface *is a pair* (P_i, P_o) *with:*

- *input ports* $P_i \subseteq \mathsf{P_i}$*, and*
- *output ports* $P_o \subseteq \mathsf{P_o}$*.*

The set of all interfaces is denoted by $\mathcal{I}$*.*

Similar to components, interfaces have an identifier which is why we postulate the existence of the set of all interface identifiers I.

Interface port valuation. As for components, the same port can be used by different interfaces. Thus, to uniquely identify an *interface port*, we need to combine it with the corresponding interface identifier. Therefore, we can generalize the notion of valuation introduced in Eq. (3) to interface ports $\mathsf{I} \times \mathsf{P}$ as done for component port valuations.

4.3 Interface Specifications

An interface specification declares a set of component and interface identifiers. Moreover, it associates an interface identifier with each component identifier and an interface with each interface identifier.

Definition 3. *An* interface specification *is a 4-tuple* (C, I, t^c, t^i) *consisting of:*

- *a set of component identifiers* $C \subseteq \mathsf{C}$*,*
- *a set of interface identifiers* $I \subseteq \mathsf{I}$*,*
- *a mapping* $t^c \colon C \to I$*, assigning an interface identifier to each component,*
- *a mapping* $t^i \colon I \to \mathcal{I}$*, which assigns an interface to each interface identifier.*

The set of all interface specifications is denoted by $\mathcal{S}_I$*.*

Convention 4. *For an n-tuple* $Z = (z_1, \ldots, z_n)$*, we denote by* $\left[z\right]^i = z_i$ *with* $1 \leqslant i \leqslant n$ *the projection to the i-th component of* Z*.*

Convention 5. *For interface specification* $S_i = (C, I, t^c, t^i) \in \mathcal{S}_I$ *we denote by:*

- $\mathsf{in}(I', S_i) \stackrel{def}{=} \bigcup_{i \in I'} (\{i\} \times \left[t^i(i)\right]^1)$ *the set of input ports,*
- $\mathsf{out}(I', S_i) \stackrel{def}{=} \bigcup_{i \in I'} (\{i\} \times \left[t^i(i)\right]^2)$ *the set of output ports,*
- $\mathsf{port}(I', S_i) \stackrel{def}{=} \mathsf{in}(I', S_i) \cup \mathsf{out}(I', S_i)$ *the set of all ports,*

for a set of interface identifiers $I' \subseteq I$*, respectively.*

The same notation can be used to denote the ports for a set of component identifiers $C' \subseteq C$ *by substituting* $t^i(i)$ *with* $t^i(t^c(c))$ *for each* $c \in C'$ *in the above definitions.*

4.4 Architecture Configurations and Configuration Traces

Architectures are modeled as sets of configuration traces which are sequences over architecture configurations.

Architecture Configurations. In our model, an architecture configuration connects ports of active components. It consists of a set of active components and a so-called connection relation connecting the component ports.

Definition 6. *An* architecture configuration *over interface specification* $S_i = (C, I, t^c, t^i) \in \mathcal{S}_I$ *is a triple* (C', N, μ)*, consisting of:*

- *a set of active components* $C' \subseteq C$*,*
- *a connection* $N \colon \underline{\mathsf{in}(C', S_i)} \dashrightarrow \wp(\mathsf{out}(C', S_i))$*,*
- *a valuation* $\mu \in \mathsf{port}(C', S_i)$*.*

We require connected ports to be consistent in their valuation, i.e. if a component provides messages at its output port, these messages are transferred to the corresponding connected input ports:

$$\forall p_i \in \mathrm{dom}\,(N) : \mu(p_i) = \bigcup_{p_o \in N(p_i)} \mu(p_o). \tag{4}$$

The set of all possible architecture configurations for interface specification $S_i \in \mathcal{S}_I$ *is denoted by* $\mathcal{K}(S_i)$*.*

Note that connection N is modeled as a set-valued, partial function from component input ports to component output ports, meaning that:

- input/output ports can be connected to several output/input ports, respectively, and
- not every input/output port needs to be connected to an output/input port, respectively.

Convention 7. *In the following we use* $c :: I$ *to denote that component variable* c *requires the corresponding component to have the assigned interface* I*. Moreover, port names are used to denote the corresponding port valuation.*

Example 1. Assuming $p_1, p_2, p_3, (p_1, s_1), (p_2, s_2) \in \mathsf{M}$, $ks_1, ks_2, bb \in \mathsf{C}$, $i_p, i_s \in \mathsf{P_i}$, and $o_p, o_s \in \mathsf{P_o}$. Figure 1 shows an architecture configuration (C', N, μ) for interface specification S_{BB} (as defined in Sect. 4.5 with $C = \{ks_1, ks_2, bb\}$), with:

- active components $C' = \{ks_1, bb\}$;
- connection N, with $N((bb, o_p)) = \{(ks_1, i_p)\}$, $N((bb, o_s)) = \{(ks_1, i_s)\}$, $N((ks_1, o_p)) = \{(bb, i_p)\}$, $N((ks_1, o_s)) = \{(bb, i_s)\}$; and
- valuation $\mu = [(ks_1, i_p), (ks_1, o_p), (bb, o_s), \cdots \mapsto \{p_1, p_2, p_3\}, \{(p_2, \{p_4\})\}, \{(p_1, s_1)\}, \cdots]$.

Convention 8. *For an architecture configuration* $k = (C', N, \mu) \in \mathcal{K}(S_i)$ *over interface specification* $S_i = (C, I, t^c, t^i) \in \mathcal{S}_I$ *we denote by*

$$\mathsf{in}^o_c(S_i, k) \quad \stackrel{def}{=} \quad \mathsf{in}(C', S_i) \setminus \mathrm{dom}\,(N)\,, \tag{5}$$

the set of open input configuration ports.

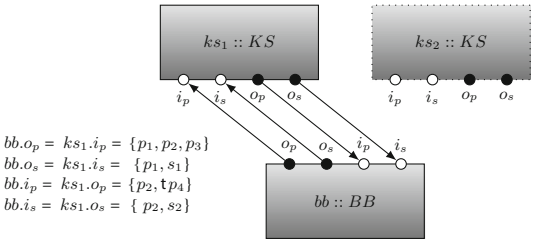

Fig. 1. Architecture configuration

Equivalences Between Architecture Configurations. Architecture configurations can be related according to several aspects. In the following we introduce several notions of architecture configuration equivalence.

Definition 9. *Two architecture configurations* $k = (C', N, \mu)$, $k' = (C'', N', \mu')$ *over interface specification* $S_i \in \mathcal{S}_I$, *with* $k, k' \in \mathcal{K}(S_i)$, *are* behavior equivalent, *written* $k \approx^b k'$, *iff*

$$\forall p \in \mathsf{port}(C' \cap C'', S_i) \colon \mu(p) = \mu'(p). \tag{6}$$

Definition 10. *Two architecture configurations* $k = (C', N, \mu)$, $k' = (C'', N', \mu')$ *over interface specification* $S_i \in \mathcal{S}_I$, *with* $k, k' \in \mathcal{K}(S_i)$, *are* activation equivalent, *written* $k \approx^a k'$, *iff*

$$C' = C''. \tag{7}$$

Definition 11. *Two architecture configurations* $k = (C', N, \mu)$, $k' = (C'', N', \mu')$ *over interface specification* $S_i \in \mathcal{S}_I$, *with* $k, k' \in \mathcal{K}(S_i)$, *are* connection equivalent, *written* $k \approx^n k'$, *iff*

$$\forall p \in \mathsf{in}(C' \cap C'', S_i) \colon N(p) = N'(p). \tag{8}$$

These relations suffice to determine architecture configuration equivalence.

Property 1. Two ACs $k, k' \in \mathcal{K}(S_i)$ are the same iff they are behavior equivalent, connection equivalent and activation equivalent:

$$k = k' \iff k \approx^b k' \wedge k \approx^n k' \wedge k \approx^a k'.$$

However, not every relation is indeed an equivalence relation.

Property 2. Activation equivalence is an equivalence relation. Behavior and connection equivalence are reflexive, symmetric, but not transitive.

Example 2 (Why behavior and connection equivalence are not necessarily transitive). Consider three architecture configurations $k' = (C'', N', \mu')$, $k = (C', N, \mu)$, $k'' = (C''', N'', \mu'') \in \mathcal{K}(S_i)$, such that $C' \subseteq C''$ and $C' \subseteq C'''$ but there exists a $c \in C'' \cap C'''$ which is not in C' and $\mu'(c, p) \neq \mu''(c, p)$ for some port p. Furthermore, assume $k' \approx^b k$ and $k \approx^b k''$. Since $\mu'(c, p) \neq \mu''(c, p)$, we have $k' \not\approx^b k''$.

A similar example can be given for connection equivalence.

Configuration traces. A configuration trace consists of a series of configuration snapshots of an architecture during system execution. Thus, a configuration trace is modeled as a sequence of architecture configurations at a certain point in time.

Definition 12. *A* configuration trace *(CT) over interface specification $S_i \in \mathcal{S}_I$ is a mapping $\mathbb{N} \to \mathcal{K}(S_i)$. The set of all CTs for S_i is denoted by $\mathcal{K}^t(S_i)$.*

Example 3. Figure 2 shows a configuration trace $t \in \mathcal{K}^t(S_i)$ with corresponding configurations $t(0) = k_0$, $t(1) = k_1$, and $t(2) = k_2$. Configuration k_0, for example, is shown in Example 1.

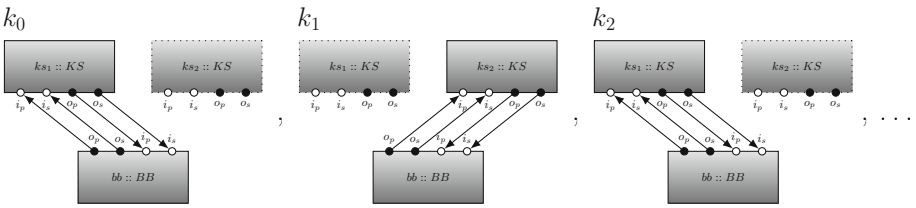

Fig. 2. Configuration trace (port valuations not shown, see Fig. 1 for an example)

Note that an architecture property is modeled as a *set of configuration traces*, rather than just one single trace. This is due to the fact that component behavior, as well as the appearance and disappearance of components, and the reconfiguration of the architecture is usually non-deterministic and dependent on the current input provided to an architecture.

Moreover, note that our notion of architecture is highly dynamic in the following sense:

- *components* may appear and disappear over time and
- *architecture configurations* may change over time.

4.5 Running Example: Blackboard Interface Specification

A Blackboard architecture consists of a `BlackBoard` component and several `KnowledgeSource` components. Figure 3 shows an interface specification $S_{BB} = (C, I, t^c, t^i) \in \mathcal{S}_I$ of the pattern.

`BlackBoard` interface. A `BlackBoard` (BB) is used to capture the current state on the way to a solution of the original problem. Its state consists of all currently open subproblems and solutions for subproblems.

A `BlackBoard` expects two types of input: 1. via i_p: a problem $p \in$ `PROB` which a `KnowledgeSource` is able to solve, together with a set of subproblems $P \subseteq$ `PROB` the `KnowledgeSource` requires to be solved before solving the original problem P, 2. via i_s: a problem $p \in$ `PROB` solved by a `KnowledgeSource`, together with the corresponding solution $s \in$ `SOL`.

A BlackBoard returns two types of output: 1. via o_p: a set $P \subseteq \texttt{PROB}$ which contains all the problems to be solved, 2. via o_s: a set of pairs $PS \subseteq \texttt{PROB} \times \texttt{SOL}$. Thus, we require the port types: $T_{i_p} = \texttt{PROB} \times \wp(\texttt{PROB})$, and $T_{i_s} = \texttt{PROB} \times \texttt{SOL}$, $T_{o_p} = \texttt{PROP}$ and $T_{o_s} = \texttt{PROB} \times \texttt{SOL}$.

KnowledgeSource interface. A KnowledgeSource (KS) is a domain expert able to solve problems in that domain. It may lack expertise of other domains. Moreover, it can recognize problems which it is able to solve and subproblems which have to be solved first by other KnowledgeSources.

A KnowledgeSource expects two types of input: 1. via i_p: a set $P \subseteq \texttt{PROB}$ which contains all the problems to be solved, 2. via i_s: a set of pairs $PS \subseteq \texttt{PROB} \times \texttt{SOL}$ containing solutions for already solved problems.

A KnowledgeSource returns one of two types of output: 1. via o_p: a problem $p \in \texttt{PROB}$ which it is able to solve together with a set of subproblems $P \subseteq \texttt{PROB}$ which it requires to be solved before solving the original problem, 2. via o_s: a problem $p \in \texttt{PROB}$ which it was able to solve together with the corresponding solution $s \in \texttt{SOL}$. Thus, we require the port types: $T_{i_p} = \texttt{PROB}$ and $T_{i_s} = \texttt{PROB} \times \texttt{SOL}$ and $T_{o_p} = \texttt{PROB} \times \wp(\texttt{PROB})$ and $T_{o_s} = \texttt{PROB} \times \texttt{SOL}$.

A KnowledgeSource can solve only certain types of problems which is why we assume the existence of a mapping $prob\colon C \to \texttt{PROB}$ to associate a set of problems with each KnowledgeSource. Then we require for each KnowledgeSource that it only solves problems given by this mapping:

$$\forall k \in \mathcal{K}(S_{BB}), (c,p) \in \texttt{out}(S_i,k)\colon t^c(c) = KS \implies \left[\left[k\right]^3(p)\right]^1 \in prob(c). \quad (9)$$

While we assume only one BlackBoard component $bb \in C$, the number of KnowledgeSource components is not restricted.

Fig. 3. Interface specification for Blackboards.

5 Specifying Properties of Dynamic Architectures

Properties of dynamic architectures can be specified as sets of configuration traces over an interface specification. In the following, we investigate the nature of such properties and introduce the notion of *behavior*, *activation*, and *connection* properties as special kinds of *architecture properties* to our model. Moreover, we introduce the notion of *separable architecture property* and show that such a

property can always be represented as the intersection of corresponding behavior, activation, and connection properties. Then, we show that the intersection of such properties is guaranteed to be non-empty, given that the properties themselves are non-empty.

This way, we get a step-wise method for the specification of properties for dynamic architectures by concentrating on the three different property-types as shown below by our running example.

5.1 Architecture Properties

We first introduce a basic notion of architecture property which serves as a foundation for all classes of architecture properties discussed below. An architecture property is a set of configuration traces which does not constrain valuation of open input ports. Thus, an architecture property is defined as a set of configuration traces fulfilling a special closure property.

Definition 13. *An* architecture property *(AP) is a set of configuration traces P, such that input port valuations are not restricted:*

$$\forall t \in P, n \in \mathbb{N}, \mu \in \overline{\mathsf{in}_c^o(S_i, t(n))}\ \exists t' \in P\colon t' \downarrow_{n-1} = t \downarrow_{n-1} \wedge$$
$$\forall p \in \mathsf{in}_c^o(S_i, t(n))\colon \left[t'(n)\right]^3(p) = \mu(p). \tag{10}$$

5.2 Behavior Properties

A behavior property is an architecture property which does not constrain connections and activations. Thus, a behavior property is defined as a set of configuration traces fulfilling a special closure property.

Definition 14. *A* behavior property (BP) *for an interface specification $S_i = (C, I, t^c, t^i) \in \mathcal{S}_I$, is an AP $B \subseteq \mathcal{K}^t(S_i)$, such that connections and activations are not restricted:*

$$\forall t \in B, n \in \mathbb{N}, k \in \{k \in \mathcal{K}(S_i) \mid k \approx^b t(n)\}$$
$$\exists t' \in B\colon t' \downarrow_{n-1} = t \downarrow_{n-1} \wedge t'(n) \approx^a k \wedge t'(n) \approx^n k. \tag{11}$$

Example 4. Figure 4 shows how an architecture property B can violate Definition 14: Assume that B allows a configuration trace t with $t(0)$ and denies some k with $k \approx^b t(0)$ at $n = 0$, i.e. $\nexists t' \in B : t'(0) \approx^a k \vee t'(n) \approx^n k$. Hence, B constrains activation and, thus, contains unnecessary parts of an activation property.

Running Example: Behavior Property Specification. We provide behavior properties for both, `BlackBoard` and `KnowledgeSource` components. Thereby we use a temporal-logic notation (based on [19]) to specify sets of configuration traces. Variables denote component identifiers, problems and solutions. Port names are used to denote port valuations and $c :: I$ is used to denote that

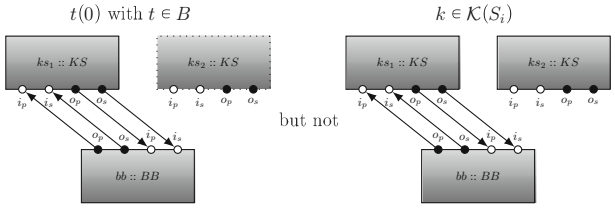

Fig. 4. Example of an ill-formed behavior property.

component identifier c has interface I.

`BlackBoard` behavior. A `BlackBoard` provides the *current state* towards solving the original problem. If a `KnowledgeSource` requires subproblems to be solved, the `BlackBoard` redirects those problems to other `KnowledgeSource`s. Moreover, the `BlackBoard` provides available solutions to all `KnowledgeSources`.

We view a `BlackBoard` as a set of configuration traces $\mathcal{K}^t(S_{BB})$ specified by three behavior properties:

– if a solution to a subproblem is received on its input, then it is eventually provided at its output:

$$\Box\left((p,s) \in (bb,i_s) \implies \Diamond\left((p,s) \in (bb,o_s)\right)\right), \tag{12}$$

– if solving a problem requires a set of subproblems to be solved first, those problems are eventually provided at its output:

$$\Box\left((p,P) \in (bb,i_p) \implies (\forall p' \in P\colon \Diamond\left(p' \in (bb,o_p)\right))\right), \tag{13}$$

– a problem is provided as long as it is not solved:

$$\Box\Big(p \in (bb,o_p) \implies \left((p \in (bb,o_p))\ \mathcal{W}\ ((p,\texttt{solve}(p)) \in (bb,i_s))\right)\Big). \tag{14}$$

`KnowledgeSource` behavior. A `KnowledgeSource` receives open problems via i_p and solutions for other problems via i_s. It might contribute to the solution of the original problem by solving subproblems. Hence, it performs one of two possible actions: 1. If it has solutions for all the required subproblems, it solves the problem and publishes the solution via o_s, 2. If it requires solutions to subproblems, it notifies the `BlackBoard` about its ability to solve the problem and about these subproblems via o_p.

We view a `KnowledgeSource` as a set of configuration traces $\mathcal{K}^t(S_{BB})$ specified by the following behavior properties:

– if a `KnowledgeSource` gets correct solutions for all the required subproblems, then it solves the problem eventually:

$$\Box\forall ks :: KS, (p,P) \in (ks,o_p)\colon \Big((\forall p \in P\colon \Diamond\left((p,\texttt{solve}(p)) \in (ks,i_s)\right)) \implies \Diamond(p,\texttt{solve}(p)) \in (ks,o_s)\Big), \tag{15}$$

– in order to solve a problem, a `KnowledgeSource` requires solutions only for smaller problems:

$$\Box\forall ks :: KS\colon\ ((p,P)\in(ks,o_p)\Longrightarrow\forall p'\in P\colon p'\prec p)\,, \tag{16}$$

– if a `KnowledgeSource` is able to solve a problem it will eventually communicate this:

$$\Box\forall ks :: KS\colon p\in prob(ks)\wedge p\in(ks,i_p)\Longrightarrow\Diamond(\exists P\subseteq\texttt{PROB}\colon (p,P)\in(ks,o_p)). \tag{17}$$

Note that Eqs. (12)-(17) constrain only the behavior of components. They do neither restrict activation nor connections. Thus, the resulting architecture property is indeed an example of a behavior property as defined in Definition 14.

5.3 Activation Properties

An architecture property is an activation property if it does neither restrict behavior nor connection. Thus, activation properties are again defined by means of a special closure property.

Definition 15. *An* activation property *(AP) for interface specification $S_i \in \mathcal{S}_I$, is an AP $A \subseteq \mathcal{K}^t(S_i)$, such that connections and behavior are not restricted:*

$$\begin{aligned}&\forall t \in A, n \in \mathbb{N}, k \in \{k \in \mathcal{K}(S_i) \mid k \approx^a t(n)\}\\ &\quad\exists t' \in A\colon t' \downarrow_{n-1} = t \downarrow_{n-1} \wedge t'(n) \approx^n k \wedge t'(n) \approx^b k.\end{aligned} \tag{18}$$

Running Example: Activation Property Specification. Activation properties of the Blackboard pattern are described in a *configuration diagram* (Fig. 5): The double solid frame for an interface (e.g. BB) denotes the condition that components have to be active from the beginning on whereas interfaces with a single frame (e.g. KS) allow components to be de-/activated over time.

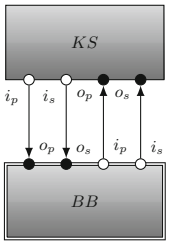

Fig. 5. Configuration diagram of Blackboards for activation and connection

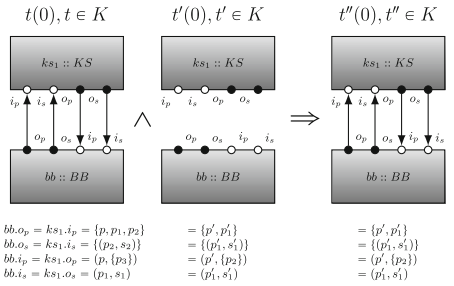

Fig. 6. Architecture violating Eq. (21)

Moreover, we require that whenever a knowledge source offers to solve some problem, it is always activated when solutions to the required subproblems are provided[1]:

$$\Box\forall c :: KS, (p, P) \in (k, o_p):$$
$$(\forall q \in P: \Diamond(q, solve(q)) \in (bb, o_s)) \implies \Diamond(q, solve(q)) \in (bb, o_s) \wedge \|c\|. \quad (19)$$

Note that the activation constraints induced by the diagram in Fig. 5 as well as Eq. (19) constrain only the activation of components. They do neither restrict connections nor behavior which is why the resulting architecture property is indeed an example of an activation property as defined in Definition 15.

5.4 Connection Properties

A connection property is not allowed to restrict neither behavior nor activation. Again this is described by a special closure property.

Definition 16. *A* connection property *(CP) for interface specification $S_i \in \mathcal{S}_I$, is an AP $N \subseteq \mathcal{K}^t(S_i)$, such that activations and behavior are not restricted:*

$$\forall t \in N, n \in \mathbb{N}, k \in \{k \in \mathcal{K}(S_i) \mid k \approx^n t(n)\}$$
$$\exists t' \in N: t' \downarrow_{n-1} = t \downarrow_{n-1} \wedge t'(n) \approx^a k \wedge t'(n) \approx^b k. \quad (20)$$

Running Example: Connection Property Specification. Connection properties are also specified graphically in the configuration diagram in Fig. 5. The solid arcs denote a constraint requiring that the ports of a `KnowledgeSource` component are connected with the corresponding ports of a `BlackBoard` component as depicted, whenever both components are active.

Note that the connection constraints induced by the diagram in Fig. 5 constrain only the connection of components. They do neither restrict activation nor behavior. Thus, the resulting architecture property is indeed an example of a connection property as defined in Definition 16.

5.5 Separable Architecture Properties

A separable architecture property is an architecture property which can be specified as the intersection of the types above.

Definition 17. *A* separable architecture property *(SAP) for interface specification $S_i = (C, I, t^c, t^i) \in \mathcal{S}_I$, is an AP $K \subseteq \mathcal{K}^t(S_i)$, such that activation, connection, and behavior do not influence each other:*

$$\forall t \in \mathcal{K}^t(S_i), n \in \mathbb{N}: \Big(\big(\exists t_b \in K: t_b \downarrow_{n-1} = t \downarrow_{n-1} \wedge t_b(n) \approx^b t(n)\big) \wedge$$
$$(\exists t_n \in K: t_n \downarrow_{n-1} = t \downarrow_{n-1} \wedge t_n(n) \approx^n t(n)) \wedge$$
$$(\exists t_a \in K: t_a \downarrow_{n-1} = t \downarrow_{n-1} \wedge t_a(n) \approx^a t(n))\Big)$$
$$\implies \exists t' \in K: t' \downarrow_n = t \downarrow_n . \quad (21)$$

[1] We use $\|c\|$ to denote that component c is active at the corresponding time.

Example 5 (Architecture violating Eq. (21)). Figure 6 shows an example of an architecture property K which violates the condition required by Eq. (21): $t''(0)$ is connection and activation equivalent with $t(0)$, and behavior equivalent with $t'(0)$. Hence, architectural property K has to permit t''.

Running Example: Blackboard Guarantee. In the following, we specify a guarantee of blackboard architectures as a separable architecture property over the interface specification S_{BB}.

Theorem 1. *Assuming that knowledge sources are active when required:*

$$\Box\Big(p \in (bb, o_p) \implies \Diamond\Big(\exists ks :: KS : p \in prob(ks) \wedge \\ (\forall p' \in P : (\Diamond((p', s) \in (bb, o_s) \implies \|ks\|)))\Big)\Big), \tag{22}$$

a Blackboard architecture guarantees to solve the original problem:

$$\Box\Big(p \in (bb, i_p) \implies \Diamond\Big((p, solve(p)) \in (bb, o_s))\Big)\Big). \tag{23}$$

Proof (Sketch. A detailed proof is given in Isabelle/HOL.). The proof is by well-founded induction over the problem relation $\prec$: We are sure that for each problem eventually a `KnowledgeSource` exists which is capable to solve the problem, Eq. (22). The required subproblems are provided to the `BlackBoard` by the connection constraint of Fig. 5. The `BlackBoard` will provide these subproblems eventually on its output o_p, Eq. (13). Since the subproblems provided to the `BlackBoard` are strictly less, Eq. (16), they will be solved and provided by the `BlackBoard` by induction over the steps 1 to 4. A `KnowledgeSource` will eventually be activated for each solution, Eq. (22), and connected to the `BlackBoard` (Fig. 5). This `KnowledgeSource` eventually has all solutions to its subproblems and will then solve the original problem by Eq. (15). The solution is received eventually by the `BlackBoard` due to Fig. 5. Finally, this solution is provided by the `BlackBoard` due to Eq. (12).

5.6 Completeness

In the following we discuss an important property of the proposed methodology which ensures that each separable architectural property can be described as the intersection of a corresponding behavior, connection, and activation property.

Theorem 2. *Each SAP $K \subseteq \mathcal{K}^t(S_i)$ for interface specification $S_i \in \mathcal{S}_I$ can be* uniquely *described through the intersection of a BP $B \subseteq \mathcal{K}^t(S_i)$, CP $N \subseteq \mathcal{K}^t(S_i)$, and AP $A \subseteq \mathcal{K}^t(S_i)$:*

$$B \cap N \cap A = K. \tag{24}$$

Proof (Sketch). Given an AP, construct the corresponding BP, AP, and CP. Then show equality of the original property and the intersection.

5.7 Consistency

Another important property of the proposed methodology regards the consistency of the different properties. It ensures that the methodology does indeed not introduce any inconsistencies. Formally, we show that the intersection of behavior, activation, and connection properties is always non-empty if the corresponding properties are non-empty.

Theorem 3. *For each BP* $B \subseteq \mathcal{K}^t(S_i)$*, CP* $N \subseteq \mathcal{K}^t(S_i)$*, and AP* $A \subseteq \mathcal{K}^t(S_i)$*, such that the properties are non-empty:*

$$B, N, A \neq \varnothing, \tag{25}$$

the intersection is non-empty: $B \cap N \cap A \neq \varnothing$*.*

Proof (Sketch). Show

$$\forall n \in \mathbb{N}\ \exists t \in \mathcal{K}^t(S_i), t_b \in B, t_n \in N, t_a \in A\colon t \downarrow_n = t_b \downarrow_n = t_a \downarrow_n = t_n \downarrow_n$$

by induction over n to have $\exists t \in B \cap N \cap A$.

6 Specifying Properties of Dynamic Architectures

In this section, we describe an approach to the specification of separable properties of dynamic architectures based on the theory discussed so far.

Properties can be specified directly by a set of configuration traces. Moreover, Fig. 7 depicts an overview of the proposed approach to separate the specification into the different types.

In a first step one has to specify an interface. Based on the interface specification one can then define behavior properties, connection properties, and activation properties. The intersection of the corresponding configuration traces represent the specified architectures.

Specifying interfaces. To specify interfaces first one has to specify a set of ports and corresponding types of messages. This can be achieved by traditional specification techniques such as algebraic specifications [30]. Interfaces can then be specified by grouping a set of ports.

Specifying behavior properties. Based on an interface specification, one can specify behavior properties. This can be achieved e.g. by specifying execution traces over the ports of an interface.

Specifying activation properties. Finally, activation properties may be specified by traces over a set of components. Such traces specify the set of active components at each point in time.

Specifying connection properties. Connection properties have to be specified as special kind of configuration traces.

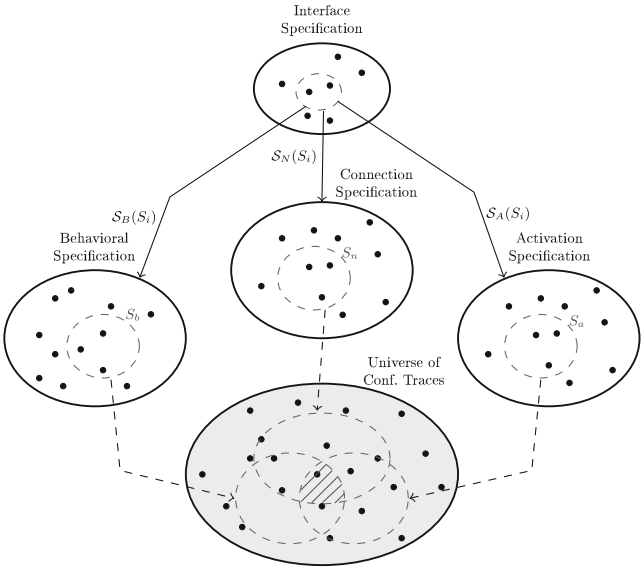

Fig. 7. Specifying Architectural Styles

Running Example: Blackboard Verification. For the verification of the blackboard architecture pattern, we transferred behavior, activation, and connection properties (see Sects. 5.2, 5.3, and 5.4) as well as the pattern's guarantee (see Sect. 5.5) into Isabelle/HOL [22]. There, we proved that each implementation complying with the three individual properties fulfills the architecture property describing the guarantee underlying any blackboard architecture[2]

7 Discussion

In the following, we briefly discuss our approach and possible limitations. Thereby, we critically examine some of its potential weaknesses in more detail.

Dynamic interfaces. One possible weakness concerns the nature of our underlying model. Definition 12 does not allow components to change their interface over time. This could be seen as a restriction of the model, however, it was a deliberate decision since for now, we did not yet find the need for components to change their interfaces. Indeed, it remains an open question whether dynamic interfaces are useful, at all. However, if the need for them arises, it should be noted, that the underlying model can be adapted to allow for dynamic interfaces as well.

[2] The script can be downloaded at http://www.marmsoler.com/pattern/Blackboard.thy

Mapping to Isabelle/HOL. Another possible weakness concerns the encoding of the blackboard example into Isabelle/HOL. The resulting Isabelle/HOL specification is indeed specific to the blackboard pattern and cannot be applied to other patterns. However, the methodology of how a pattern specification can be systematically translated into a corresponding Isabelle/HOL specification is indeed generalizable to other patterns as well. Indeed, the mapping could be fully automated for specifications in our language.

Quality attributes. A last point which needs to be discussed in more detail regards an important aspect of software architectures in general. Our approach does actually not provide means to directly specify quality attributes such as performance, availability, etc. However, as our example shows, it allows us to specify the technical realization of such aspects. The theorem provided for the blackboard pattern ensures, that a problem can be solved also in the absence of certain components. This can be actually seen as one possible implementation (or technical definition) of what is sometimes called reliability.

8 Conclusion

In this article, we provide a formal notion of properties for dynamic architectures and investigate different types of properties. The major results can be summarized as follows:

- We provide a *novel model for dynamic architectures* and a formal notion of *properties* for these kind of architectures (Sect. 4). Thereby we introduce the notion of architecture configuration and configuration traces (Definition 6). Then we model an architecture property as a set of configuration traces (Definition 12) for which open input port valuations are not restricted (Definition 13).
- We provide a characterization of *behavior properties*, *activation properties*, and *connection properties* for dynamic architectures (Sect. 5). Each property-type is defined as an architecture property fulfilling a special closure property: A behavior property is not allowed to restrict activations or connections (Definition 14). An activation property, on the other hand, is neither allowed to restrict connections nor behavior (Definition 15). Finally, a connection property is not allowed to restrict activation or behavior (Definition 16).
- We provide a characterization of *separable architecture properties*, architecture properties which can be separated into behavior, activation, and connection parts (Definition 17). We show that each separable architecture property can indeed be separated into behavior, activation, and connection properties (Theorem 2). We show that the intersection of behavior, activation, and connection properties always yields a non-empty architecture property (Theorem 3).

We evaluated our results by deriving a systematic way to specify properties for dynamic architectures and apply it to the specification of blackboard architectures (Sect. 6):

– We specified the constraints imposed by the pattern as behavior, activation, and connection properties.
– We formulated the pattern's guarantee as an architecture property.
– We verified the correctness of the pattern by proving its guarantee from the pattern's constraints in Isabelle/HOL.

We imagine the following implications of our results: (i) The proposed model can be used to specify properties for dynamic architectures. (ii) The results on the different types of properties provide a systematic way to specify separable architecture properties for those kinds of architectures by focusing on different aspects of a dynamic architecture.

We perceive future work in three major areas:

– Based on the insights provided by our results, we aim to build specialized specification and analysis techniques for the three identified property-types: activation, connection, and behavior properties. Especially the specification of behavior properties remains an open issue since they are usually specified locally to a component instead of over the whole architecture. Thus, we are currently investigating how such local specifications can be transformed to specifications over dynamic architectures where the specified component can be activated/deactivated over time.
– Another direction of work concerns the transformation of specifications to (interactive) theorem provers to support the verification of specifications. Currently we are working on a systematic way to transform specifications in the presented formalism to corresponding Isabelle/HOL specifications.
– Finally, the approach should be applied to specify and investigate patterns for dynamic architectures to further evaluate our approach and (maybe even more important) to provide detailed insights into the nature of existing patterns as well as to discover new patterns for dynamic architectures.

Acknowledgments. We would like to thank Jonas Eckhardt, Vasileios Koutsoumpas, and the reviewers of ICTAC 2016 for their comments and helpful suggestions.

References

1. Abowd, G.D., Allen, R., Garlan, D.: Formalizing style to understand descriptions of software architecture. ACM TOSEM **4**, 319–364 (1995)
2. Allen, R., Douence, R., Garlan, D.: Specifying and analyzing dynamic software architectures. In: Astesiano, E. (ed.) FASE 1998. LNCS, vol. 1382, pp. 21–37. Springer, Heidelberg (1998). doi:10.1007/BFb0053581
3. Allen, R.J.: A formal approach to software architecture. Technical report, DTIC Document (1997)
4. Bernardo, M., Ciancarini, P., Donatiello, L.: On the formalization of architectural types with process algebras. ACM SIGSOFT SEN **25**, 140–148 (2000)
5. Bradbury, J.S., Cordy, J.R., Dingel, J., Wermelinger, M.: A survey of self-management in dynamic software architecture specifications. In: WOSS (2004)
6. Broy, M.: A logical basis for component-oriented software and systems engineering. Comput. J. **53**(10), 1758–1782 (2010)

7. Broy, M.: A model of dynamic systems. In: Bensalem, S., Lakhneck, Y., Legay, A. (eds.) ETAPS 2014. LNCS, vol. 8415, pp. 39–53. Springer, Heidelberg (2014). doi:10.1007/978-3-642-54848-2_3
8. Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., Stal, M.: A system of patterns: Pattern-oriented software architecture (1996)
9. Castro, P.F., Aguirre, N.M., López Pombo, C.G., Maibaum, T.S.E.: Towards managing dynamic reconfiguration of software systems in a categorical setting. In: Cavalcanti, A., Deharbe, D., Gaudel, M.-C., Woodcock, J. (eds.) ICTAC 2010. LNCS, vol. 6255, pp. 306–321. Springer, Heidelberg (2010). doi:10.1007/978-3-642-14808-8_21
10. Clements, P.C.: A survey of architecture description languages. In: IWSSD (1996)
11. Dashofy, E.M., Van der Hoek, A., Taylor, R.N.: A highly-extensible, xml-based architecture description language. In: WICSA, IEEE (2001)
12. Fiadeiro, J.L., Lopes, A.: A model for dynamic reconfiguration in service-oriented architectures. Softw. Syst. Model. **12**(2), 349–367 (2013)
13. Garlan, D.: Formal modeling and analysis of software architecture: components, connectors, and events. In: Bernardo, M., Inverardi, P. (eds.) SFM 2003. LNCS, vol. 2804, pp. 1–24. Springer, Heidelberg (2003). doi:10.1007/978-3-540-39800-4_1
14. Hirsch, D., Montanari, U.: Two graph-based techniques for software architecture reconfiguration. Electron. Notes Theor. Comput. Sci. **51**, 177–190 (2002)
15. Inverardi, P., Wolf, A.L.: Formal specification and analysis of software architectures using the chemical abstract machine model. IEEE TSE **21**, 373–386 (1995)
16. Le Métayer, D.: Describing software architecture styles using graph grammars. IEEE TSE **24**, 521–533 (1998)
17. Luckham, D.C., Kenney, J.J., Augustin, L.M., Vera, J., Bryan, D., Mann, W.: Specification and analysis of system architecture using Rapide. IEEE TSE **21**, 336–355 (1995)
18. Magee, J., Kramer, J.: Dynamic structure in software architectures. ACM SIGSOFT SEN **21**, 3–14 (1996)
19. Manna, Z., Pnueli, A.: The Temporal Logic of Reactive and Concurrent Systems: Specification. Springer, New york (2012)
20. Medvidovic, N.: ADLs and dynamic architecture changes. In: ISAW (1996)
21. Moriconi, M., Qian, X., Riemenschneider, R.A.: Correct architecture refinement. IEEE TSE **21**, 356–372 (1995)
22. Nipkow, T., Paulson, L.C., Wenzel, M.: Isabelle/HOL: A Proof Assistant for Higher-Order Logic. Springer Science & Business Media, Heidelberg (2002)
23. Oquendo, F.: π-ADL: an architecture description language based on the higher-order typed π-calculus for specifying dynamic and mobile software architectures. ACM SIGSOFT SEN **29**, 1–14 (2004)
24. Penix, J., Alexander, P., Havelund, K.: Declarative specification of software architectures. In: ASE (1997)
25. Shaw, M., Garlan, D.: Software Architecture: Perspectives on an Emerging Discipline, vol. 1. Prentice Hall Englewood Cliffs, Upper Saddle River (1996)
26. Spivey, J.M., Abrial, J.: The Z notation (1992)
27. Taylor, R.N., Medvidovic, N., Dashofy, E.M.: Software Architecture: Foundations, Theory, and Practice. Wiley Publishing, Hoboken (2009)
28. Wenzel, M.: Isabelle/Isar: a generic framework for human-readable proof documents. From Insight to Proof: Festschrift in Honour of Andrzej Trybulec **10**, 277–298 (2007)
29. Wermelinger, M., Lopes, A., Fiadeiro, J.L.: A graph based architectural (re) configuration language. ACM SIGSOFT SEN **26**(5), 21–32 (2001)
30. Wirsing, M.: Algebraic Specification. MIT Press, Cambridge (1991)