Noname manuscript No.
(will be inserted by the editor)

On the Specification of Constraints for Dynamic Architectures

Diego Marmsoler

Received: date / Accepted: date

Abstract In dynamic architectures, component activation and connections between components may vary over time. With the emergence of mobile computing such architectures became increasingly important and several techniques emerged to support in their specification. These techniques usually allow for the specification of concrete architecture instances. Sometimes, however, it is desired to focus on the specification of constraints, rather than concrete architectures. Especially specifications of architecture patterns usually focus on a few, important constraints, leaving out the details of the concrete architecture implementing the pattern. With this article we introduce an approach to specify such constraints for dynamic architectures. To this end, we introduce the notion of *configuration traces* as an abstract model for dynamic architectures. Then, we introduce the notion of *configuration trace assertions* as a formal language based on linear temporal logic to specify constraints for such architectures. In addition, we also introduce the notion of *configuration diagrams* to specify interfaces and certain common activation and connection constraints in one single, graphical notation. The approach is well-suited to specify patterns for dynamic architectures and verify them by means of formal analyses. This is demonstrated by applying the approach to specify and verify the Blackboard pattern for dynamic architectures.

Keywords Dynamic Architectures · Algebraic Specifications · Architecture Constraints

Diego Marmsoler
Technische UniversitÃďt MÃijnchen
E-mail: diego.marmsoler@tum.de

1 Introduction

A systems *architecture* describes the components of a system as well as connections between those components. *Dynamic architectures* are architectures in which component activation as well as connections can change over time [35]. Specifying such architectures remains an active topic of research [15,8] and over the last years, several approaches emerged to support in this endeavor [22,3,14,8]. These approaches usually focus on the specification of concrete architecture instances characterized by the following properties:

- *Concrete model of execution*: Component behavior is either specified using some notion of state-machine [2], guarded commands [35], or even stream-processing functions [8].
- *Concrete model of component interaction*: Interaction between components is either message-synchronous (for approaches based on CSP [18]), time-synchronous [8], or also action-synchronous [35].
- *Concrete model of component activation/deactivation*: Component activation/deactivation is either specified by arbitrary components [22] or by a designated component [2].
- *Concrete model of reconfiguration*: Similarly, *connection changes* are either implemented by each single component [22] or again by a designated component [2].

While these aspects are important when specifying concrete architecture instances, they play only a secondary role when specifying architectural constraints as it is the case when specifying architectural patterns, for example.

Consider, for example, the specification of the Blackboard architecture pattern. In this pattern, a set of experts (aka. Knowledge-sources) collaborate through a so-called Blackboard component to collaboratively solve a complex problem [10,31,33]. The pattern requires that for each problem provided by a Blackboard component, eventually an expert exists which is able to handle this problem. In turn, the pattern guarantees that a complex problem can be collaboratively solved, even if no single expert exists which can solve the problem on its own.

If we look at this specification, several observations can be made: (i) The specification does not prescribe how a component is implemented as long as it satisfies certain behavioral constraints. (ii) Neither does it constrain how components communicate as long as they do. (iii) Moreover, it is not specified who creates a component as long as it is created somehow. (iv) Finally, it is not specified how the connections were established as long as they are. Thus, we argue, that traditional architecture specification techniques meet its limits when it comes to the specification of such constraints and that new, more abstract techniques are required.

To address this problem, we introduce an abstract model of dynamic architectures. Thereby, an architecture is modeled as a set of so-called configuration traces which are sequences over architecture configurations. An architecture configuration, on the other hand, consists of a set of active components, connections between component ports and (important) a valuation of the component ports with messages.

Based on this model, we provide a model-theoretic approach to formally specify properties of dynamic architectures by means of architecture constraints: (i) First, abstract datatypes are specified by means of algebraic specifications. (ii) Then, interfaces and component types are specified over these datatypes by means of interface specifications. (iii) Finally, properties can be specified over the interfaces by means of configuration trace assertions. To facilitate the specification process we also introduce the notion of configuration diagrams as a graphical means to specify interfaces as well as certain common activation and connection constraints.

The approach allows to specify properties of dynamic architectures by means of architecture constraints and is thus well suited for the specification of patterns for this kind of architectures. To demonstrate this, we evaluate the approach by specifying and verifying the *Blackboard* pattern [10,31,33] for dynamic architectures. Thereby, we identify and formalize the patterns key architecture constraints as a set of configuration trace assertions and prove its guarantee from the specification.

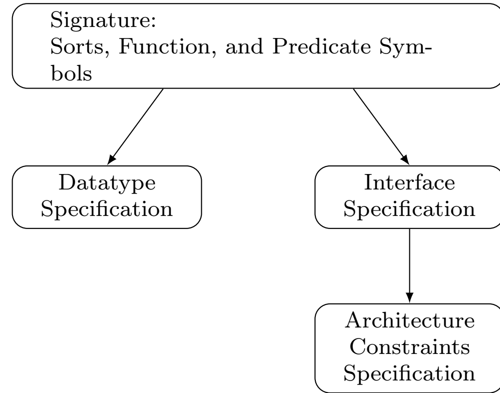

Figure 1 Approach to specify properties of dynamic architectures.

1.1 Specifying Properties of Dynamic Architectures

Fig. 1 provides an overview of our approach to specify properties of dynamic architectures. As a first step, a suitable signature is specified to introduce symbols for sets, functions, and predicates. These symbols form the primitive entities of the whole specification process: datatype specifications and interface specifications as well as architecture constraint specifications are based on these symbols.

Then, datatypes are specified over the signature. Therefore, so-called datatype assertions are build over datatype terms to assert the datatypes characteristic properties and provide meaning for the symbols introduced in the signature.

Interfaces are also directly specified over the signature. Therefore, a set of ports is typed by sorts of the corresponding signature. Then, an interface is specified by assigning an interface identifier with three sets of ports: local, input, and output ports. Finally, a set of interface assertions is associated with each interface identifier to specify component types, i.e., interfaces with associated global invariants.

Finally, architecture constraints can be specified by means of configuration trace assertions over the interfaces. Configuration trace assertions are a special kind of linear-temporal formulas build over configuration assertions, i.e., assertions over an architecture configuration.

1.2 Running Example: Blackboard Architectures

In the following we introduce the Blackboard architecture pattern. It is used as a running example throughout the text to illustrate the main concepts and ideas.

The Blackboard pattern (as described, for example, by Shaw and Garlan [31], Buschmann et al. [10], and

Taylor et al. [33]), is a pattern for dynamic architectures used for collaborative problem solving. In a Blackboard architecture, several experts (aka. Knowledge-sources) collaborate through a central component (aka. Blackboard) to solve a complex problem consisting of several sub-problems.

Although the pattern is not too complex (consisting of only two types of components), it incorporates several aspects of dynamic architectures: (i) Knowledge-source components can be activated and deactivated over time, (ii) connections between the various Knowledge-sources and the Blackboard component can also change over time.

1.3 Overview

The remainder of this article is structured as follows: In Sect. 2 we discuss our model of dynamic architectures. Then, the different modeling techniques to specify datatypes (Sect. 3), interfaces (Sect. 4), and configuration traces (Sect. 5) are introduced. For each technique we provide a formal description of its syntax as well as its semantics in terms of the model introduced in Sect. 2. While these techniques already suffice to specify all kinds of architecture properties, in Sect. 6 we introduce the notion of configuration diagrams as a graphical notation to support in the specification of interfaces and certain common activation and connection constraints. In Sect. 7 we demonstrate how a specification in our language can be used to formally reason about the specification. To this end, we verify the Blackboard architecture pattern by proving one of its characteristic properties from its specification. Finally, we discuss our approach and possible limitations thereof in Sect. 8. In Sect. 9 we provide related work and conclude with a brief summary in Sect. 10.

To not lose track of the various concepts and notations, Sect. 2 – Sect. 6 conclude with a tabular overview of all the new concepts and notations introduced in the corresponding section.

Sect. 12 provides our notation for some general, mathematical concepts. If at any time during the reading a symbol was used but not properly introduced, its definition can be found in this section.

2 A Model of Dynamic Architectures

In the following, we describe our model of dynamic architectures introduced in [25]. It is based on Broy's Focus theory [7] and an adaptation of its dynamic extension [8]. An architecture property is thereby modeled as a set of configuration traces which are sequences of architecture configurations that, in turn, consist of a set of active components, valuations of their ports with messages, and connections between their ports.

2.1 Foundations

In our model, components communicate by exchanging messages over ports. Thus, we assume the existence of sets M and P containing all messages and ports, respectively.

2.1.1 Port-Valuations

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.

Definition 1 (Port-valuation) For a set of ports $P \subseteq \mathsf{P}$, we denote by $\overline{P}$ the set of all possible *port-valuations*, formally:

$$\overline{P} \quad \stackrel{\text{def}}{=} \quad (P \to \wp(\mathsf{M})) \ .$$

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.

2.1.2 Components

In our model, the basic unit of computation is a component. It consists of an identifier, a set of ports and a valuation of ports with messages. Indeed it is rather a snapshot of a component at a certain point in time with concrete messages on its ports.

Thus, we assume the existence of set C_{id} containing all component identifiers.

Definition 2 (Component) A *component* is a 5-tuple (d, L, I, O, μ) consisting of:

- a component identifier $d \in \mathsf{C}_{id}$;
- *disjoint* sets of local ports $L \subseteq \mathsf{P}$, input ports $I \subseteq \mathsf{P}$, and output ports $O \subseteq \mathsf{P}$; and
- a valuation of its ports $\mu \in \overline{L \cup I \cup O}$.

The set of all components is denoted by $\mathcal{C}$.

For a set of components $C \subseteq \mathcal{C}$, we denote by:

- $\mathsf{loc}(C) \stackrel{\text{def}}{=} \bigcup_{(d,L,I,O,\mu)\in C}(\{d\}\times L)$ the set of *component local ports*,
- $\mathsf{in}(C) \stackrel{\text{def}}{=} \bigcup_{(d,L,I,O,\mu)\in C}(\{d\}\times I)$ the set of *component input ports*,

– $\mathsf{out}(C) \stackrel{\text{def}}{=} \bigcup_{(d,L,I,O,\mu)\in C}(\{d\} \times O)$ the set of *component output ports*,
– $\mathsf{port}(C) \stackrel{\text{def}}{=} \mathsf{loc}(C) \cup \mathsf{in}(C) \cup \mathsf{out}(C)$ the set of all *component ports*, and
– $\mathsf{id}(C) \stackrel{\text{def}}{=} \bigcup_{(d,L,I,O,\mu)\in C}(\{d\})$ the set of all *component identifiers*.

A set of components $C \subseteq \mathcal{C}$ is called *healthy* if for each (d, L, I, O, μ), $(d', L', I', O', \mu') \in C$ the following conditions are fulfilled:

– a component's interface is determined by its identifier:

$$d = d' \implies L = L' \land I = I' \land O = O' \ , \tag{1}$$

– the valuation of the local ports is also determined by a component's identifier:

$$d = d' \implies \forall p \in L \colon \mu(p) = \mu'(p) \ . \tag{2}$$

Note 1 (Well-definedness of Eq. (2)) Due to Eq. (1) $L = L'$ which is why Eq. (2) is indeed well-defined.

A healthy set of components $C \subseteq \mathcal{C}$ induces mappings to assigning local, input, output, and all ports to component identifier $d \in \mathsf{id}(C)$:

– $\mathsf{loc}_C(d) = O \iff \exists I, O \subseteq \mathsf{P} \colon (d, L, I, O) \in C$,
– $\mathsf{in}_C(d) = I \iff \exists O, L \subseteq \mathsf{P} \colon (d, L, I, O) \in C$,
– $\mathsf{out}_C(d) = O \iff \exists I, L \subseteq \mathsf{P} \colon (d, L, I, O) \in C$, and
– $\mathsf{port}_C(d) \stackrel{\text{def}}{=} \mathsf{loc}_C(d) \cup \mathsf{in}_C(d) \cup \mathsf{out}_C(d)$.

Moreover, it induces a mapping lV_C to access the valuation of a component's local ports:

$$lV_C(d)(p) = M \iff \exists I, O, L \subseteq \mathsf{P}, \mu \in \overline{L \cup I \cup O} \colon \\ (d, L, I, O, \mu) \in C \land \mu(p) = M \ . \tag{3}$$

Note 2 (Well-definedness of $\mathsf{loc}_C, \mathsf{in}_C, \mathsf{out}_C$ and $lV_C(d)$) While Eq. (1) guarantees that loc_C, in_C, and out_C are well-defined, Eq. (2) guarantees that $lV_C(d)$ is well-defined.

Note that a healthy set of components does restrict only the interface of components and the port-valuations of its local ports. However, there may be several components with the same identifier but different valuations of its input and output ports. Thus, it is indeed possible to have two *different* components $(d, L, I, O, \mu), (d, L, I, O, \mu')$ in a healthy set of components, as long as there exists a port $p \in I \cup O$, such that $\mu(p) \neq \mu'(p)$.

An important property of healthy is, that it is preserved under the subset relation.

Property 1 (Subset preserves healthiness) For a healthy set of components C, each subset $C' \subseteq C$ is again healthy.

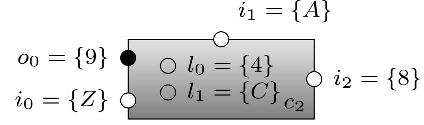

Figure 2 Conceptual representation of a Component with identifier c_2, local ports l_0, l_1, input ports i_0, i_1, i_2, output ports o_0, and corresponding valuations $\{4\}, \{C\}, \{Z\}, \{A\}, \{8\}, \{9\}$.

Proof Assume C is healthy and let $C' \subseteq C$. Moreover, let $c = (d, L, I, O, \mu), c' = (d', L', I', O', \mu') \in C'$. We first show $d = d' \implies L = L' \land I = I' \land O = O'$. Thus, assume $d = d'$ and have $L = L' \land I = I' \land O = O'$ by Eq. (1), since $c, c' \in C$. Now we show $d = d' \implies \forall p \in L \colon \mu(p) = \mu'(p)$. Thus, assume $d = d'$, let $p \in L$ and have $\mu(p) = \mu'(p)$ by Eq. (2), since $c, c' \in C$.

Example 1 (Component) Assuming M consists of all characters and numbers, $c_2 \in \mathsf{C}_{id}$, and $l_0, l_1, i_0, i_1, i_2, o_0 \in \mathsf{P}$. Figure 2 shows a conceptual representation of a component (d, L, I, O, μ), with:

– identifier: $d = c_2$,
– local ports: $L = \{l_0, l_1\}$,
– input ports: $I = \{i_0, i_1, i_2\}$,
– output ports: $O = \{o_0\}$, and
– valuation μ defined as follows:
  – $\mu(l_0) = \{4\}$ and $\mu(l_1) = \{C\}$,
  – $\mu(i_0) = \{Z\}$, $\mu(i_1) = \{A\}$, and $\mu(i_2) = \{8\}$; and
  – $\mu(o_0) = \{9\}$.

2.2 Architecture Configurations and Configuration Traces

Architecture properties are modeled as sets of *configuration traces* which are sequences over *architecture configurations*.

2.2.1 Architecture Configurations

In our model, an architecture configuration *connects* ports of *active* components.

Definition 3 (Architecture configuration) An *architecture configuration* over a *healthy* set of components $C \subseteq \mathcal{C}$ is a pair (C', N), consisting of:

– a set of active components $C' \subseteq C$ and
– a connection of their ports $N \colon \mathsf{in}(C') \dashrightarrow \wp(\mathsf{out}(C'))$.

We require the valuation of active components of an architecture configuration to be determined by a component's identifier:

$$\forall (d, L, I, O, \mu), (d', L', I', O', \mu') \in C' \colon \\ d = d' \implies \mu = \mu'. \tag{4}$$

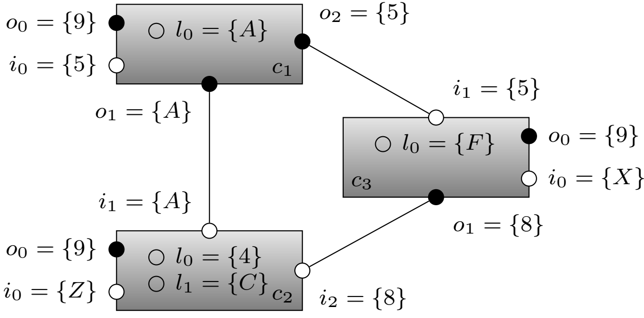

Figure 3 Architecture configuration consisting of 3 components and a connection between ports (c_2, i_1) and (c_1, o_1), (c_2, i_2) and (c_3, o_1), and (c_3, i_1) and (c_1, o_2).

Thus, we can define a function to obtain the valuation for a component in an architecture configuration by means of its identifier, characterized by the following equation:

$$val_{(C',N)}(d) = \mu \iff \exists L,I,O \subseteq \mathsf{P} \colon (d,L,I,O,\mu) \in C' \quad . \tag{5}$$

Note 3 (Well-definedness of function $val_{(C',N)}$) Function $val_{(C',N)}$ is well-defined by Eq. (5), due to Eq. (4).

Moreover, we require connected ports to be consistent in their valuation, that is, if a component provides messages at its output ports, these messages are transferred to the corresponding, connected input ports:

$$\forall (d_i, p_i) \in \operatorname{dom}(N) \colon val_{(C',N)}(d_i)(p_i) = \bigcup_{(d_o,p_o) \in N(d_i,p_i)} val_{(C',N)}(d_o)(p_o) \quad . \tag{6}$$

The set of all possible architecture configurations over a healthy set of components $C \subseteq \mathcal{C}$ is denoted by $\mathcal{K}(C)$.

Note that a connection 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.

Example 2 (Architecture configuration) Assuming M consists of all characters and numbers, $c_1, c_2, c_3 \in \mathsf{C}_{id}$, and $l_0, l_1, i_0, i_1, i_2, o_0, o_1, o_2 \in \mathsf{P}$. Figure 3 shows an architecture configuration (C', N), with:

- active components $C' = \{C_1, C_2, C_3\}$, where C_2, for example, is shown in Ex. 1, and
- connection N defined as follows:
  - $N((c_2, i_1)) = \{(c_1, o_1)\}$,
  - $N((c_3, i_1)) = \{(c_1, o_2)\}$, and
  - $N((c_2, i_2)) = \{(c_3, o_1)\}$.

Moreover, due to the healthiness condition, an active component within an architecture configuration is fully determined by its identifier.

Property 2 For a healthy set of components $C \subseteq \mathcal{C}$, we have

$$\forall (C', N) \in \mathcal{K}(C), \forall c, c' \in C' \colon [c]^1 = [c']^1 \implies c = c' \quad . \tag{7}$$

Proof Assume $c = (d, L, I, O, \mu)$ and $c = (d', L', I', O', \mu')$ and $d = d'$. Then, by Eq. (1), we have that $L = L'$, $I = I'$, and $O = O'$. Moreover, by Eq. (4), we have $\mu = \mu'$. Thus, we can conclude $c = c'$. □

2.2.2 Configuration Traces

A configuration trace consists of a series of configuration snapshots of an architecture during system execution.

Definition 4 (Configuration trace) A *configuration trace* over a healthy set of components $C \subseteq \mathcal{C}$ is a mapping $\mathbb{N} \to \mathcal{K}(C)$. The set of all configuration traces over C is denoted by $\mathcal{R}(C)$.

Example 3 (Configuration trace) Figure 4 shows a conceptual representation of a configuration trace $t \in \mathcal{R}(C)$ with corresponding architecture configurations $t(0) = k_0$, $t(1) = k_1$, and $t(2) = k_2$. Architecture configuration t_0, e.g., is shown in Ex. 2.

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 input to an architecture is usually nondeterministic and the appearance and disappearance of components, as well as the reconfiguration of an architecture, may indeed depend on the input provided to it.

Moreover, note that our notion of architecture is dynamic in the following sense: (i) *components* may appear and disappear over time and (ii) *connections* may change over time.

2.3 Summary

Table 1 provides a brief overview of the main concepts introduced in this section. For each concept it provides a brief description thereof and related notation.

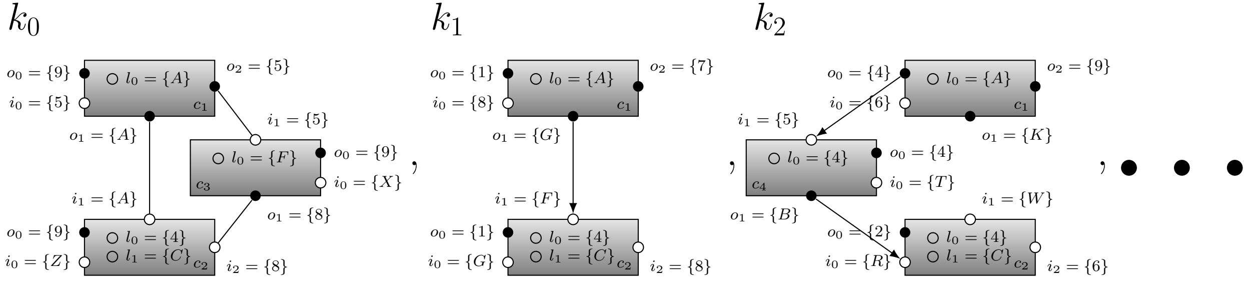

Figure 4 First three architecture configurations k_0, k_1, and k_2, of a configuration trace.

Table 1 Overview of concepts for dynamic architectures.

Concept	*Description*	*Related Notation*
message	atomic data entity	M
port	means to exchange messages	P
port-valuation	assignment of messages to set of ports P	$\overline{P}$
component identifier	identifier for components	C_{id}
component	identifier, local/input/output ports, and port-valuations *local, input, output, all ports of components C* *identifiers of components C*	$\mathcal{C}$ $\mathsf{loc}(C)$, $\mathsf{in}(C)$, $\mathsf{out}(C)$, $\mathsf{port}(C)$ $\mathsf{id}(C)$
healthy set of components	set of components with interface and valuations of local ports determined by component identifiers *local, input, output, and all ports of component with identifier d of healthy set C* *valuation of local ports of component with identifier d of healthy set C*	 $\mathsf{loc}_C(d), \mathsf{in}_C(d), \mathsf{out}_C(d), \mathsf{port}_C(d)$ $lV_C(d)$
architecture configuration	set of active components and connections between their ports over a healthy set of components C *valuation of component with identifier d of architecture configuration k*	$\mathcal{K}(C)$ $val_k(d)$
configuration trace	sequence of architecture configurations over a set of healthy components C	$\mathcal{R}(C)$

3 Datatype Specifications

Datatypes are specified by means of algebraic specifications [6,36].

Thus, a datatype specification is expressed over a signature by means of a set of so-called datatype assertions, i.e., predicate-logic formulas over datatype terms. Meaning is provided in terms of a corresponding algebra, i.e., concrete mathematical structures for the sorts and functions of the corresponding signature.

3.1 Signatures

A signature determines the symbols used throughout the specification. Sorts are symbols representing certain sets of messages while function symbols and predicate symbols, represent functions, and predicates over those sets.

Definition 5 (Signature) A *signature* is a triple $\Sigma = (S, F, B)$, consisting of:

- a set of sorts S,
- a set of function symbols F and predicate symbols B with corresponding assignments $sort\colon F \to S^n$ and $sort\colon B \to S^n$, with:
  - F^n/B^n denoting the set of function/predicate symbols with arity $n \in \mathbb{N}$,
  - $sort_n(f)/sort_n(b)$ denoting the sort of the n-th parameter (with $n \in \mathbb{N}^+$) of function symbol $f \in F$ / predicate symbol $b \in B$, and
  - $sort_0(f)$ denoting the sort of the return value of function symbol $f \in F$.

3.2 Algebras

The meaning of the symbols introduced by a signature is determined by an algebra. An algebra consists of

concrete sets for each sort-symbol and corresponding functions and predicates for the function-symbols and predicate-symbols, respectively. Moreover, mappings associate each symbol with the corresponding interpretation.

Definition 6 (Algebra) An *algebra* for a signature (S, F, B) is a 6-tuple $(S', F', B', \alpha, \beta, \gamma)$, consisting of:

- a set of *non-empty* sets of messages $S' \subseteq \wp(\mathsf{M})$;
- a set of functions F' and predicates B' for symbols F and B, respectively; and
- interpretations $\alpha\colon S \to S'$, $\beta\colon F \to F'$, and $\gamma\colon B \to B'$.

The set of all possible algebras over signature Σ is denoted by $\mathcal{A}(\Sigma)$.

3.3 Datatype Terms

Terms of datatype specifications are build over a given signature and corresponding *datatype variables* (a family of disjoint sets of variables ${}_d\mathcal{V} = ({}_d\mathcal{V}_s)_{s\in S}$ with ${}_d\mathcal{V}_s$ denoting a set of variables of sort $s \in S$).

Definition 7 (Datatype term) The set of all *datatype terms* of sort $s \in S$ over a signature $\Sigma = (S, F, B)$ and datatype variables ${}_d\mathcal{V}$ is the smallest set ${}^s_d\mathcal{T}(\Sigma, {}_d\mathcal{V})$, satisfying the equations in Fig. 5. The set of all datatype terms (of all sorts) over a signature $\Sigma = (S, F, B)$ and datatype variables ${}_d\mathcal{V}$ is denoted by ${}_d\mathcal{T}(\Sigma, {}_d\mathcal{V})$.

Roughly speaking, a datatype term is the application of function symbols of a signature to other function symbols or variables. Thereby, the sorts of the parameters have to be consistent with the sorts of the corresponding function symbol.

The semantics of datatype terms is defined over an algebra $A = (S', F', B', \alpha, \beta, \gamma)$ and corresponding *datatype variable assignment* (a family of mappings $\iota = (\iota_s)_{s\in S}$ with $\iota_s\colon {}_d\mathcal{V}_s \to \alpha(s)$ for each sort $s \in S$). In the following we denote with $\mathcal{I}_A^{{}_d\mathcal{V}}$ the set of all datatype variable assignments for datatype variables ${}_d\mathcal{V}$ in algebra A.

Definition 8 (Datatype semantic function) The *datatype semantic function* for datatype terms ${}_d\mathcal{T}(\Sigma, {}_d\mathcal{V})$ with signature $\Sigma = (S, F, B)$ and datatype variables ${}_d\mathcal{V}$, over algebra $A \in \mathcal{A}(\Sigma)$ and datatype variable assignment $\iota \in \mathcal{I}_A^{{}_d\mathcal{V}}$ is the mapping $[\![_]\!]^\iota_A\colon {}^s_d\mathcal{T}(\Sigma, {}_d\mathcal{V}) \to \alpha(s)$ (for each sort $s \in S$), characterized by the equations in Fig. 6.

Thus, the semantics of a datatype term is given by a function assigning a value of the corresponding algebra to each term.

3.4 Datatype Assertions

Datatype assertions are build over datatype terms by the common logical operators.

Definition 9 (Datatype assertion) The set of all *datatype assertions* over a signature Σ and datatype variables ${}_d\mathcal{V}$ is the smallest set ${}_d\Gamma(\Sigma, {}_d\mathcal{V})$ satisfying the equations in Fig. 7.

Thus, a datatype assertion is obtained by applying the common logical operators to datatype terms or predicate symbols. The semantics of datatype assertions is defined over an algebra.

Definition 10 (Datatype models relation) The *datatype models relation* for datatype assertions ${}_d\Gamma(\Sigma, {}_d\mathcal{V})$ with signature Σ, datatype variables ${}_d\mathcal{V}$, and algebra $A \in \mathcal{A}(\Sigma)$ is the relation $A, _ \models _ \subseteq \mathcal{I}_A^{{}_d\mathcal{V}} \times {}_d\Gamma(\Sigma, {}_d\mathcal{V})$ characterized by the equations in Fig. 8. A datatype assertion $\varphi \in {}_d\Gamma(\Sigma, {}_d\mathcal{V})$ is *valid* for an algebra $A \in \mathcal{A}(\Sigma)$ iff there exists a datatype variable assignment $\iota \in \mathcal{I}_A^{{}_d\mathcal{V}}$, such that $A, \iota \models \varphi$. An algebra $A \in \mathcal{A}(\Sigma)$ is a *model* for datatype assertion $\varphi \in {}_d\Gamma(\Sigma, {}_d\mathcal{V})$, written $A \models \varphi$ iff $A, \iota \models \varphi$ for each $\iota \in \mathcal{I}_A^{{}_d\mathcal{V}}$. An algebra $A \in \mathcal{A}(\Sigma)$ is a model for a set of datatype assertions $\Phi \subseteq {}_d\Gamma(\Sigma, {}_d\mathcal{V})$, written $A \models \Phi$ iff $A \models \varphi$ for each $\varphi \in \Phi$.

Thus, the semantics is given by a relation over datatype assertions and algebras with a corresponding datatype variable assignment satisfying the assertions.

3.5 Specifying Datatypes

Signatures introduce the basic symbols used throughout the whole specification process and datatype specifications provide meaning to these symbols.

Definition 11 (Datatype specification) A datatype specification over a signature $\Sigma = (S, F, B)$ and a family of datatype variables ${}_d\mathcal{V} = ({}_d\mathcal{V}_s)_{s\in S}$ is a set of datatype assertions $\Phi \subseteq {}_d\Gamma(\Sigma, {}_d\mathcal{V})$.

Signatures and corresponding datatype specifications can be expressed by means of datatype specification templates (Fig. 9). Each template has a name and can import other datatype specification templates by means of their name. Sorts are introduced by a list of names at the beginning of the template. Then, a list of variables for the different sorts are defined and function/predicate symbols are introduced with the corresponding types. Finally, a list of datatype assertions is specified to describe the characteristic properties of a datatype.

$$
\begin{aligned}
v \in {}_d\mathcal{V}_s &\implies v \in {}^{s}_{d}\mathcal{T}(\Sigma, {}_d\mathcal{V}) \ , \\
f \in F^0 &\implies f \in {}^{s}_{d}\mathcal{T}(\Sigma, {}_d\mathcal{V}) \text{ [for } sort_0(f) = s] \ , \\
f \in F^{n+1} \wedge t_1 \in {}^{s_1}_{d}\mathcal{T}(\Sigma, {}_d\mathcal{V}), \cdots, t_{n+1} \in {}^{s_{n+1}}_{d}\mathcal{T}(\Sigma, {}_d\mathcal{V}) &\implies f(t_1, \cdots, t_{n+1}) \in {}^{s}_{d}\mathcal{T}(\Sigma, {}_d\mathcal{V}) \text{ [for } n \in \mathbb{N}, sort_0(f) = s, \\
&\qquad \text{and } sort_1(f) = s_1, \cdots, sort_{n+1}(f) = s_{n+1}] \ .
\end{aligned}
$$

Figure 5 Inductive definition of datatype terms ${}^{s}_{d}\mathcal{T}(\Sigma, {}_d\mathcal{V})$ of sort $s \in S$ over signature $\Sigma = (S, F, B)$ and datatype variables ${}_d\mathcal{V} = ({}_d\mathcal{V}_s)_{s \in S}$.

$$
\begin{aligned}
[\![v]\!]^{\iota}_{A} &\overset{\text{def}}{=} \iota_s(v) \text{ [for } v \in {}_d\mathcal{V}_s] \ , \\
[\![f]\!]^{\iota}_{A} &\overset{\text{def}}{=} \beta(f) \text{ [for function symbol } f \in F^0] \ , \\
[\![f(t_1, \cdots, t_n)]\!]^{\iota}_{A} &\overset{\text{def}}{=} \beta(f)([\![t_1]\!]^{\iota}_{A}, \cdots, [\![t_n]\!]^{\iota}_{A}) \text{ [for function symbol } f \in F^{n+1}] \ .
\end{aligned}
$$

Figure 6 Recursive definition of datatype semantic function for datatype terms ${}^{s}_{d}\mathcal{T}(\Sigma, {}_d\mathcal{V})$ of sort $s \in S$ with signature $\Sigma = (S, F, B)$, algebra $A = (S', F', B', \alpha, \beta, \gamma) \in \mathcal{A}(\Sigma)$, and datatype variable assignment $\iota = (\iota_s)_{s \in S}$.

$$
\begin{aligned}
b \in B^0 &\implies b \in {}_d\Gamma(\Sigma, {}_d\mathcal{V}) \ , \\
b \in B^{n+1} \wedge t_1 \in {}^{s_1}_{d}\mathcal{T}(\Sigma, {}_d\mathcal{V}), \cdots, t_{n+1} \in {}^{s_{n+1}}_{d}\mathcal{T}(\Sigma, {}_d\mathcal{V}) &\implies b(t_1, \cdots, t_{n+1}) \in {}_d\Gamma(\Sigma, {}_d\mathcal{V}) \\
&\qquad \text{[for } n \in \mathbb{N} \text{ and } sort_1(b) = s_1, \cdots, sort_{n+1}(b) = s_{n+1}] \ , \\
t, t' \in {}^{s}_{d}\mathcal{T}(\Sigma, {}_d\mathcal{V}) &\implies t = t' \in {}_d\Gamma(\Sigma, {}_d\mathcal{V}) \ , \\
E \in {}_d\Gamma(\Sigma, {}_d\mathcal{V}) &\implies \neg E \in {}_d\Gamma(\Sigma, {}_d\mathcal{V}) \ , \\
E, E' \in {}_d\Gamma(\Sigma, {}_d\mathcal{V}) &\implies E \wedge E', \ E \vee E', \ E \implies E', \ E \iff E' \in {}_d\Gamma(\Sigma, {}_d\mathcal{V}) \ , \\
E \in {}_d\Gamma(\Sigma, {}_d\mathcal{V}) \wedge x \in {}_d\mathcal{V}_s &\implies \forall x.\ E \in {}_d\Gamma(\Sigma, {}_d\mathcal{V}) \ \wedge \\
&\qquad \exists x.\ E \in {}_d\Gamma(\Sigma, {}_d\mathcal{V}) \text{ [for some } s \in S] \ .
\end{aligned}
$$

Figure 7 Inductive definition of datatype assertions ${}_d\Gamma(\Sigma, {}_d\mathcal{V})$ over signature $\Sigma = (S, F, B)$ and datatype variables ${}_d\mathcal{V} = ({}_d\mathcal{V}_s)_{s \in S}$.

DTSpec Name	**imports** dtSpec
sort `Sort1`, `Sort2`	
var var1, var2 : `Sort1` var3 : `Sort2`	
symbol1 : `Sort1` symbol2 : `Sort1` → `Sort2`	
assertion1(symbol1, symbol2, var1, var2, var4) assertion2(symbol1, symbol2, var1, var2, var4)	

Figure 9 Datatype specification template with corresponding sorts, datatype variables, symbols for functions and predicates, and datatype assertions.

3.6 Blackboard: Datatype Specification

Blackboard architectures work with *problems* and *solutions* for these problems. Figure 10 provides the corresponding datatype specification template. We denote with `PROB` the set of all problems and with `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* $\prec \subseteq$ `PROB` $\times$ `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[1] (Eq. (8)) 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 certain problem (i.e. several ways to split a problem into subproblems), we assume (without loss of generality) that the final solution for a problem is always unique. Thus, we assume the existence of a function *solve*: `PROB` $\rightarrow$ `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.

[1] A partial order is well-founded iff it does not contain any infinite decreasing chains. A detailed definition can be found e.g. in [19].

$$
\begin{aligned}
A,\iota \models b &\iff \gamma(b) \text{ [for } b \in B^0] \ , \\
A,\iota \models b(t_1,\cdots,t_n) &\iff \gamma(b)([\![t_1]\!]^\iota_A,\cdots,[\![t_n]\!]^\iota_A) \text{ [for } b \in B^{n+1}] \ , \\
A,\iota \models t = t' &\iff [\![t]\!]^\iota_A = [\![t']\!]^\iota_A \ , \\
A,\iota \models E \wedge E' &\iff A,\iota \models E \wedge A,\iota \models E' \ , \\
A,\iota \models E \vee E' &\iff A,\iota \models E \vee A,\iota \models E' \ , \\
A,\iota \models E \implies E' &\iff A,\iota \models E \implies A,\iota \models E' \ , \\
A,\iota \models E \iff E' &\iff A,\iota \models E \iff A,\iota \models E' \ , \\
A,\iota \models \exists x.\ E &\iff \exists x' \in \alpha(s)\colon A,\iota[s\colon x \mapsto x'] \models E \text{ [for } s \in S \text{ and } x \in {}_d\mathcal{V}_s] \ , \\
A,\iota \models \forall x.\ E &\iff \forall x' \in \alpha(s)\colon A,\iota[s\colon x \mapsto x'] \models E \text{ [for } s \in S \text{ and } x \in {}_d\mathcal{V}_s] \ .
\end{aligned}
$$

Figure 8 Recursive definition of models relation for datatype assertions ${}_d\Gamma(\Sigma, {}_d\mathcal{V})$ with signature $\Sigma = (S, F, B)$, algebra $A = (S', F', B', \alpha, \beta, \gamma) \in \mathcal{A}(\Sigma)$, and datatype variable assignment $\iota = (\iota_s)_{s \in S}$.

Table 2 Overview of concepts for datatype specifications.

Concept	*Description*	*Related Notation*
signature	sorts, function/predicate symbols *n-ary function/predicate symbols* *sort of n-th parameter of function symbol f / predicate symbol b* *sort of return value of function symbol f*	Σ F^n/B^n $sort_n(f)$ / $sort_n(b)$ $sort_0(f)$
algebra	sets, functions, predicates, and corresponding mappings for a signature Σ	$\mathcal{A}(\Sigma)$
datatype variable	variable for datatype elements of sort s	${}_d\mathcal{V}_s$
datatype variable assignment	assignment of elements of an algebra A to a set of datatype variables ${}_d\mathcal{V}$	$\mathcal{I}^{{}_d\mathcal{V}}_A$
datatype term	term over a signature Σ and datatype variables ${}_d\mathcal{V}$	${}^s_d\mathcal{T}(\Sigma, {}_d\mathcal{V})$, ${}_d\mathcal{T}(\Sigma, {}_d\mathcal{V})$
datatype semantic function	assigns elements of an algebra A to datatype terms under a certain datatype variable assignment ι	$[\![_]\!]^\iota_A$
datatype assertion	formula over datatype terms with corresponding signature Σ and datatype variables ${}_d\mathcal{V}$	${}_d\Gamma(\Sigma, {}_d\mathcal{V})$
datatype models relation	relates datatype assertions with algebras and corresponding datatype variable assignment ι	$_,\iota \models _$ / $_ \models _$
datatype specification	set of datatype assertions	Φ
datatype specification template	structured technique to specify datatypes	graphical

DTSpec ProbSol		**imports** SET
sort PROB, SOL		
$\prec$:	PROB $\times$ PROB	
solve:	PROB $\rightarrow$ SOL	
$well-founded(\prec)$		(8)

Figure 10 Blackboard datatype specification template introducing sorts, function symbols, and predicate symbols for Blackboard architectures.

3.7 Summary

To conclude, Tab. 2 provides a brief overview of the main concepts introduced in this section. For each concept it provides a brief description and related notation.

4 Interface Specifications

Interfaces are specified over a given signature and declare a set of local, input, and output ports for a set of interface identifiers. Moreover, an interface specification allows to specify valuations of local ports by means of interface assertions formulated over interface terms.

Thus, in the following, we postulate the existence of the set of all *port identifiers* P_{id}.

4.1 Port Specifications and Interfaces

Ports are specified by means of port specifications which declare a set of port identifiers and a corresponding typing.

Definition 12 (Port specification) A *port specification* over signature $\Sigma = (S, F, B)$ is a pair (P, t^p), consisting of:

- a set of port identifiers $P \subseteq \mathsf{P}_{id}$ and
- a mapping $t^p \colon P \to S$ assigning a sort to each port identifier.

The set of all port specifications over signature Σ is denoted by $\mathcal{S}_p(\Sigma)$.

Interfaces are build over a given port specification. They consist of a set of local, input, and output port identifiers.

Definition 13 (Interface) An *interface* over port specification $(P, t^p) \in \mathcal{S}_p(\Sigma)$ is a triple (L_i, I_i, O_i), consisting of *disjoint* sets for:

- local port identifiers $L_i \subseteq P$,
- input port identifiers $I_i \subseteq P$, and
- output port identifiers $O_i \subseteq P$.

The set of all interfaces over port specification S_p is denoted by $\mathcal{I}(S_p)$.

An interface can be interpreted by a component, relating port identifiers of the interface with concrete ports of the component.

Definition 14 (Interface interpretation) An *interface interpretation* for an interface $(L_i, I_i, O_i) \in \mathcal{I}(S_p)$ over port specification $S_p = (P, t^p) \in \mathcal{S}_p(\Sigma)$ with signature Σ in an algebra $A = (S', F', B', \alpha, \beta, \gamma)$ is a 4-tuple $(c, \delta^l, \delta^i, \delta^o)$, consisting of:

- a component $c = (d, L, I, O, \mu) \in \mathcal{C}$, and
- port interpretations $\delta^l \colon L \leftrightarrow L_i$, $\delta^i \colon I \leftrightarrow I_i$, and $\delta^o \colon O \leftrightarrow O_i$, for local, input, and output ports, respectively.

Thereby, we require that the valuations of the component ports satisfy the typing constraints induced by the corresponding port specification:

$$\begin{aligned}
&\forall p \in L \colon \mu(p) \in \alpha(t^p(\delta^l(p))), \\
&\forall p \in I \colon \mu(p) \in \alpha(t^p(\delta^i(p))), \text{and} \\
&\forall p \in O \colon \mu(p) \in \alpha(t^p(\delta^o(p))) \ .
\end{aligned} \tag{9}$$

The set of all interface interpretations of interface Q under algebra A is denoted by $\mathcal{Q}(Q, A)$.

4.2 Interface Terms

Interface terms are build over a given interface, corresponding signature, and datatype variables.

Definition 15 (Interface term) The set of all *interface terms* of sort $s \in S$ of signature $\Sigma = (S, F, B)$ over an interface $Q \in \mathcal{I}(S_p)$ with $S_p \in \mathcal{S}_p(\Sigma)$ and datatype variables ${}_d\mathcal{V}$ is the smallest set ${}^s_i\mathcal{T}_{{}_d\mathcal{V}}(\Sigma, Q)$ satisfying the equations of Fig. 11. The set of all interface terms of all sorts is denoted by ${}_i\mathcal{T}_{{}_d\mathcal{V}}(\Sigma, Q)$.

The semantics of interface terms is defined over a given algebra, corresponding datatype variable assignment, and interface interpretation. It is given in terms of a function assigning a value of the corresponding algebra to each interface term.

Definition 16 (Interface semantic function) The *interface semantic function* for *interface terms* ${}_i\mathcal{T}_{{}_d\mathcal{V}}(\Sigma, Q)$ over signature $\Sigma = (S, F, B)$ and interface Q in algebra $A = (S', F', B', \alpha, \beta, \gamma) \in \mathcal{A}(\Sigma)$ with corresponding datatype variable assignment $\iota \in \mathcal{I}_A^{{}_d\mathcal{V}}$, and interface interpretation $j = (c, \delta^l, \delta^i, \delta^o) \in \mathcal{Q}(Q, A)$ is the function $[\![_]\!]^j_{(A,\iota)} \colon {}^s_i\mathcal{T}_{{}_d\mathcal{V}}(\Sigma, Q) \to \alpha(s)$ (for each sort $s \in S$), characterized by the equations in Fig. 12.

4.3 Interface Assertions

Interface assertions are build by the common logical operators over interface terms. They are formulated over a given interface and datatype variables.

Definition 17 (Interface assertion) The set of all interface assertions over a signature Σ, interface Q, and datatype variables ${}_d\mathcal{V}$ is the smallest set ${}_i\Gamma_{{}_d\mathcal{V}}(\Sigma, Q)$ satisfying the equations in Fig. 13.

The semantics of interface assertions is given by relating interface assertions with corresponding interface interpretations satisfying the assertions.

Definition 18 (Interface models relation) The *interface models relation* for *interface assertions* ${}_i\Gamma_{{}_d\mathcal{V}}(\Sigma, Q)$ over signature Σ, datatype variables ${}_d\mathcal{V}$, and interface $Q \in \mathcal{I}(S_p)$ with $S_p \in \mathcal{S}_p(\Sigma)$ in algebra $A \in \mathcal{A}(\Sigma)$ with corresponding datatype variable assignment $\iota \in \mathcal{I}_A^{{}_d\mathcal{V}}$ is the relation $_ \models^\iota_A _ \subseteq \mathcal{Q}(Q, A) \times {}_i\Gamma_{{}_d\mathcal{V}}(\Sigma, Q)$ characterized by the equations in Fig. 14. An interface assertion $\gamma \in {}_i\Gamma_{{}_d\mathcal{V}}(\Sigma, Q)$ is *valid* for algebra $A \in \mathcal{A}(\Sigma)$ and interface interpretation $j \in \mathcal{J}(Q, A)$ iff there exists a corresponding datatype variable assignment $\iota \in \mathcal{I}_A^{{}_d\mathcal{V}}$, such that $j \models^\iota_A \gamma$. Interface interpretation $j \in \mathcal{J}(Q, A)$ is a *model* for $\gamma \in {}_i\Gamma_{{}_d\mathcal{V}}(\Sigma, Q)$, written $j \models_A \gamma$ iff for each corresponding datatype variable

$$
\begin{aligned}
v \in {}_d\mathcal{V}_s &\implies v \in {}^s_i\mathcal{T}_{{}_d\mathcal{V}}(\Sigma, Q) \ , \\
f \in F^0 &\implies f \in {}^s_i\mathcal{T}_{{}_d\mathcal{V}}(\Sigma, Q) \ [\text{for } \mathit{sort}_0(f) = s] \ , \\
f \in F^{n+1} \land t_1 \in {}^{s_1}_i\mathcal{T}_{{}_d\mathcal{V}}(\Sigma, Q), \cdots, t_{n+1} \in {}^{s_{n+1}}_i\mathcal{T}_{{}_d\mathcal{V}}(\Sigma, Q) &\implies f(t_1, \cdots, t_{n+1}) \in {}^s_i\mathcal{T}_{{}_d\mathcal{V}}(\Sigma, F) \ [\text{for } n \in \mathbb{N},\ \mathit{sort}_0(f) = s, \\
&\qquad \text{and } \mathit{sort}_1(f) = s_1, \cdots, \mathit{sort}_{n+1}(f) = s_{n+1}] \ , \\
p \in [Q]^1 \cup [Q]^2 \cup [Q]^3 &\implies p \in {}^s_i\mathcal{T}_{{}_d\mathcal{V}}(\Sigma, Q) \ [\text{if } t^p(p) = s] \ .
\end{aligned}
$$

Figure 11 Inductive definition of interface terms ${}^s_i\mathcal{T}_{{}_d\mathcal{V}}(\Sigma, Q)$ of sort $s \in S$ over signature $\Sigma = (S, F, B)$, interface Q, and datatype variables ${}_d\mathcal{V}$.

$$
\begin{aligned}
[\![v]\!]^j_{(A,\iota)} &\stackrel{\text{def}}{=} \iota_s(v) \ [\text{for } v \in {}_d\mathcal{V}_s] \ , \\
[\![f]\!]^j_{(A,\iota)} &\stackrel{\text{def}}{=} \beta(f) \ [\text{for function symbol } f \in F^0] \ , \\
[\![f(t_1, \cdots, t_n)]\!]^j_{(A,\iota)} &\stackrel{\text{def}}{=} \beta(f)([\![t_1]\!]^\iota_A, \cdots, [\![t_n]\!]^\iota_A) \ [\text{for function symbol } f \in F^{n+1}] \ , \\
[\![p]\!]^j_{(A,\iota)} &\stackrel{\text{def}}{=} \mu((\delta^i)^{-1}(p)) \ [\text{for } p \in I_i] \ , \\
[\![p]\!]^j_{(A,\iota)} &\stackrel{\text{def}}{=} \mu((\delta^o)^{-1}(p)) \ [\text{for } p \in O_i] \ .
\end{aligned}
$$

Figure 12 Recursive definition of interface semantic function for interface terms ${}^s_i\mathcal{T}_{{}_d\mathcal{V}}(\Sigma, Q)$ of sort $s \in S$ with signature $\Sigma = (S, F, B)$, interface $Q = (L_i, I_i, O_i)$, datatype variables ${}_d\mathcal{V}$, algebra $A = (S', F', B', \alpha, \beta, \gamma) \in \mathcal{A}(\Sigma)$ and corresponding datatype variable assignments $\iota \in \mathcal{I}^{{}_d\mathcal{V}}_A$, and interface interpretation $j = (c, \delta^l, \delta^i, \delta^o) \in \mathcal{Q}(Q, A)$.

$$
\begin{aligned}
b \in B^0 &\implies b \in {}_i\Gamma_{{}_d\mathcal{V}}(\Sigma, Q) \ , \\
b \in B^{n+1} \land t_1 \in {}^{s_1}_i\mathcal{T}_{{}_d\mathcal{V}}(\Sigma, Q), \cdots, t_{n+1} \in {}^{s_{n+1}}_i\mathcal{T}_{{}_d\mathcal{V}}(\Sigma, Q) &\implies b(t_1, \cdots, t_{n+1}) \in {}_i\Gamma_{{}_d\mathcal{V}}(\Sigma, Q) \\
&\qquad [\text{for } n \in \mathbb{N} \text{ and } \mathit{sort}_1(b) = s_1, \cdots, \mathit{sort}_{n+1}(b) = s_{n+1}] \ , \\
t, t' \in {}_c\mathcal{T}^{{}_c\mathcal{V}}_{{}_d\mathcal{V}}(\Sigma, S_i) &\implies t = t' \in {}_i\Gamma_{{}_d\mathcal{V}}(\Sigma, Q) \ , \\
E \in {}_i\Gamma_{{}_d\mathcal{V}}(\Sigma, Q) &\implies \neg E \in {}_i\Gamma_{{}_d\mathcal{V}}(\Sigma, Q) \ , \\
E, E' \in {}_i\Gamma_{{}_d\mathcal{V}}(\Sigma, Q) &\implies E \land E',\ E \lor E',\ E \implies E',\ E \iff E' \in {}_i\Gamma_{{}_d\mathcal{V}}(\Sigma, Q) \ , \\
E \in {}_i\Gamma_{{}_d\mathcal{V}}(\Sigma, Q) \land x \in {}_d\mathcal{V}_s &\implies \forall x.\ E \in {}_i\Gamma_{{}_d\mathcal{V}}(\Sigma, Q) \ \land \\
&\qquad \exists x.\ E \in {}_i\Gamma_{{}_d\mathcal{V}}(\Sigma, Q) \ [\text{for some } s \in S] \ .
\end{aligned}
$$

Figure 13 Inductive definition of interface assertions ${}_i\Gamma_{{}_d\mathcal{V}}(\Sigma, Q)$ over signature $\Sigma = (S, F, B)$, interface Q, and datatype variables ${}_d\mathcal{V}$.

$$
\begin{aligned}
j \models^\iota_A b &\iff \gamma(b) \ [\text{for } b \in B^0] \ , \\
j \models^\iota_A b(t_1, \cdots, t_n) &\iff \gamma(b)([\![t_1]\!]^\iota_A, \cdots, [\![t_n]\!]^\iota_A) \ [\text{for } b \in B^{n+1}] \ , \\
j \models^\iota_A t = t' &\iff [\![t]\!]^\iota_A = [\![t']\!]^\iota_A \ , \\
j \models^\iota_A E \land E' &\iff j \models^\iota_A E \land j \models^\iota_A E' \ , \\
j \models^\iota_A E \lor E' &\iff j \models^\iota_A E \lor j \models^\iota_A E' \ , \\
j \models^\iota_A E \implies E' &\iff j \models^\iota_A E \implies j \models^\iota_A E' \ , \\
j \models^\iota_A E \iff E' &\iff j \models^\iota_A E \iff j \models^\iota_A E' \ , \\
j \models^\iota_A \exists x.\ E &\iff \exists x' \in \alpha(s) \colon j \models^{\iota[s:\ x \mapsto x']}_A E \ [\text{for } s \in S \text{ and } x \in {}_d\mathcal{V}_s] \ , \\
j \models^\iota_A \forall x.\ E &\iff \forall x' \in \alpha(s) \colon j \models^{\iota[s:\ x \mapsto x']}_A E \ [\text{for } s \in S \text{ and } x \in {}_d\mathcal{V}_s] \ .
\end{aligned}
$$

Figure 14 Recursive definition of models relation for interface assertions ${}_i\Gamma_{{}_d\mathcal{V}}(\Sigma, Q)$ over signature $\Sigma = (S, F, B)$, interface Q, and datatype variables ${}_d\mathcal{V}$ over algebra $A \in \mathcal{A}(\Sigma)$ with corresponding datatype variable assignment $\iota = (\iota_s)_{s \in S}$.

assignment $\iota \in \mathcal{I}_A^{d\mathcal{V}}$ we have $j \models_A^\iota \gamma$. Interface interpretation $j \in \mathcal{J}(Q, A)$ is a *model* for a set of interface assertions $\Gamma \subseteq {}_i\Gamma_{d}\mathcal{V}(\Sigma, Q)$, written $j \models_A \Gamma$ iff $j \models_A \gamma$ for each $\gamma \in \Gamma$.

4.4 Specifying Interfaces

Interfaces are specified by providing a set of interface identifiers. Then, each identifier is associated with an interface (i.e., sets of local, input and output ports). Finally, a sets of interface assertions is specified for each interface identifier.

Thus, in the following, we postulate the existence of the set of all *interface identifiers* I_{id}.

Definition 19 (Interface specification) An *interface specification* over port specification $S_p \in \mathcal{S}_p(\Sigma)$ with signature Σ is a pair (N, Q), consisting of:

- a set of interface identifiers $N \subseteq \mathsf{I}_{id}$,
- a family of corresponding interfaces $(Q_i)_{i\in N}$ with interface $Q_i \in \mathcal{I}(S_p)$ for each interface identifier $i \in N$.

The set of all interface specifications over port specification $S_p \in \mathcal{S}_p(\Sigma)$ is denoted by $\mathcal{S}_i(S_p)$.

For an interface specification $S_i = (N, Q)$, we denote by:

- $\mathsf{loc}(S_i) \stackrel{\text{def}}{=} \bigcup_{i\in N}(i \times [Q_i]^1)$ the set of *interface local ports*,
- $\mathsf{in}(S_i) \stackrel{\text{def}}{=} \bigcup_{i\in N}(i \times [Q_i]^2)$ the set of *interface input ports*,
- $\mathsf{out}(S_i) \stackrel{\text{def}}{=} \bigcup_{i\in N}(i \times [Q_i]^3)$ the set of *interface output ports*, and
- $\mathsf{port}(S_i) \stackrel{\text{def}}{=} \mathsf{loc}(S_i) \cup \mathsf{in}(S_i) \cup \mathsf{out}(S_i)$ the set of all *interface ports.*

Note that an interface specification actually specifies a set of interfaces, rather than just one single interface. Moreover, it allows for reuse of ports through several interfaces. Thus, if a port is specified once, it can be used to specify several, different interfaces.

Interface specifications can be interpreted by a set of components with corresponding interfaces.

Definition 20 (Interface specification interpretation) An *interface specification interpretation* for an interface specification $(N, Q) \in \mathcal{S}_i(S_p)$ over port specification $S_p \in \mathcal{S}_p(\Sigma)$ under an algebra $A = (S', F', B', \alpha, \beta, \gamma) \in \mathcal{A}(\Sigma)$ is a family $J = (J_i)_{i\in N}$, with $J_i \subseteq \mathcal{Q}(Q_i, A)$ being the biggest set of interface interpretations for interface identifier $i \in N$, such that

PSpec Name		**uses** dtSpec
p:	`Sort1`	
q:	`Sort2`	
c:	`Sort3`	

Figure 15 Port specification template defining three ports and corresponding sorts.

- components with the same identifier belong only to one interface:

$$\Big(\bigcap_{i\in N} \bigcup_{(c,\delta^l,\delta^i,\delta^o)\in J_i} \{[c]^1\}\Big) = \emptyset \ . \tag{10}$$

- the set of all components $\bigcup_{i\in N} \bigcup_{(c,\delta^l,\delta^i,\delta^o)\in J_i} \{c\}$ is a healthy set of components.

We introduce a function to return the set of all components for a certain interface identifier:

$$cmp_i(J) = \bigcup_{(c,\delta^l,\delta^i,\delta^o)\in J_i} \{c\} \ . \tag{11}$$

With $cmp(J) = \bigcup_{i\in N} cmp_i(J)$ we denote the set of all components of all interface identifiers.

Note that $cmp(J)$ is a healthy set of component which allows us to apply all the functions for healthy sets of components introduced in Def. 2 to $cmp(J)$. The set of all possible interface specification interpretations for interface specification S_i over algebra A is denoted by $\mathcal{J}(S_i, A)$.

4.4.1 From Interfaces To Component Types

Remember the healthiness condition (Eq. (2)) requiring local port valuations not to change for components. Once fixed, those ports do not change their value during the execution of an architecture and are thus a way to parametrize components. The annotated interfaces become component types since they enrich an interface with certain semantic constraints.

The parametrization step is done during interface specification since interface assertions are the only way to determine/use local port valuations. It is demonstrated in the running example for Knowledge-source components which are parametrized by a set of problems they are able to solve.

4.4.2 Specification Using Templates

Component types can be specified by means of port specification templates and interface specification templates. Fig. 15, for example, shows a port specification templates declaring 3 ports and corresponding sorts. Fig. 16 shows a corresponding interface specification

ISpec Name	**based on** pSpec **uses** dtSpec
loc: c *in:* p *out:* q	
p:	`Sort1`
var aVar :	`Sort4`
$someAxiom(p, q, c, aVar)$	

Figure 16 Interface specification template consisting of a port specification and definitions for variables as well as interface assertions.

template. Each specification has a name with corresponding input, output, and local ports and may use some datatype specifications. Then, a list of variables and corresponding sorts is specified. Finally, a list of interface assertions can be specified over the ports and variables.

Sometimes it is convenient to combine a port and interface specification into a single interface specification template. Fig. 16, e.g., declares an additional port p and corresponding sort.

4.5 Blackboard: Interface Specification

A Blackboard architecture consists of a Blackboard component and several Knowledge-source components.

4.5.1 Blackboard Interface

A Blackboard (BB) component 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 already solved subproblems.

A BB expects two types of input: 1. a problem $p \in \texttt{PROB}$ which a Knowledge-source is able to solve, together with a set of subproblems $P \subseteq \texttt{PROB}$ the Knowledge-source requires to be solved before solving the original problem p, 2. a problem $p \in \texttt{PROB}$ solved by a Knowledge-source, together with the corresponding solution $s \in \texttt{SOL}$. A BB returns two types of output: 1. a set $P \subseteq \texttt{PROB}$ which contains all the problems to be solved, 2. a set of pairs $PS \subseteq \texttt{PROB} \times \texttt{SOL}$ containing solved problems and the corresponding solutions.

4.5.2 Knowledge-Source Interface

A Knowledge-source (KS) component 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 KSs.

PSpec Blackboard	**uses** ProbSol
r_p :	$\texttt{PROB} \times \wp(\texttt{PROB})$
n_s :	$\texttt{PROB} \times \texttt{SOL}$
o_p :	`PROP`
c_s :	$\texttt{PROB} \times \texttt{SOL}$

Figure 17 Blackboard port specification.

ISpec BB	**based on** Blackboard
loc: *in:* r_p, n_s *out:* $o_p c_s$	

Figure 18 Blackboard interface specification.

ISpec KS	**based on** Blackboard **uses** ProbSol
loc: prob *in:* o_p, c_s *out:* r_p, n_s	
prob:	$\wp(\texttt{PROB})$
var p :	`PROP`
P :	$\wp(\texttt{PROP})$
$r_p = (p, P) \implies p \in prob$	(12)

Figure 19 Knowledge source interface specification.

A KS expects two types of input: 1. a set $P \subseteq \texttt{PROB}$ which contains all the problems to be solved, 2. a set of pairs $PS \subseteq \texttt{PROB} \times \texttt{SOL}$ containing solutions for already solved problems. A KS returns one of two types of output: 1. 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. a problem $p \in \texttt{PROB}$ which it was able to solve together with the corresponding solution $s \in \texttt{SOL}$.

Figure 17 shows a port specification template of the Blackboard pattern.

Based on the induced port specification, Fig. 18 shows the corresponding interface specification for the pattern.

A KS can only solve certain types of problems which is why we assume the existence of a local port *prob* for each knowledge source which is typed by the set of problems a certain KS can solve. In Eq. (12) we require for each KS that it only solves problems given by this mapping.

4.6 Summary

To conclude this section, Tab. 3 provides a brief overview of the main concepts introduced in this section. For each concept it provides a brief description and related notation.

Table 3 Overview of concepts for interface specifications.

Concept	*Description*	*Related Notation*
port identifier	identifier for ports	P_{id}
port specification	port identifiers and corresponding typing w.r.t. a signature Σ	$\mathcal{S}_p(\Sigma)$
interface	set of local, input, and output port identifiers of a port specification P	$\mathcal{I}(P)$
interface interpretation	components and corresponding port interpretations for an interface F	$\mathcal{Q}(F,)$
port interpretation	bijection between port identifiers and ports of a component	δ^l, δ^i ,δ^o
interface term	term over the ports of a certain interface F, signature Σ, and datatype variables ${}_d\mathcal{V}$	${}_i\mathcal{T}_{{}_d\mathcal{V}}(\Sigma, F)$
interface semantic function	assigns elements of an algebra A to interface terms under datatype variable assignment ι and interface interpretation J	$[\![_]\!]^J_{(A,\iota)}$
interface assertion	formula over interface terms with signature Σ, interface F, and datatype variables ${}_d\mathcal{V}$	${}_i\Gamma_{{}_d\mathcal{V}}(\Sigma, F)$
interface models relation	relates interface assertions φ with interface interpretation J under algebra A and datatype variable assignment ι	$J \models^\iota_A \varphi / J \models_A \varphi$
interface identifier	identifier for interfaces	I_{id}
interface specification	interface-identifiers and corresponding interfaces over signature Σ *local, input, output, and all ports of interface spec* S_i	$\mathcal{S}_i(\Sigma)$ $\mathsf{loc}(S_i)$, $\mathsf{in}(S_i)$, $\mathsf{out}(S_i)$, $\mathsf{port}(S_i)$
interface specification interpretation	set of interface interpretations for interfaces of an interface specification S_i under algebra A	$\mathcal{J}(S_i, A)$
port specification template	structured technique to specify ports	graphical
interface specification template	structured technique to specify component types	graphical

5 Architecture Constraint Specification

Architecture constraints are specified as temporal logic formulas over architecture configurations. Thus, we first introduce the notion of *configuration assertion* to specify architecture configurations. Then, we introduce *configuration trace assertions* as an extension of configuration assertions to specify configuration traces.

5.1 Configuration Assertions

Architecture configurations can be specified by so-called configuration assertions formulated over configuration terms.

5.1.1 Configuration Terms

Terms of configuration assertions are build over an interface specification, corresponding signature, datatype variables and *component variables* (a family of disjoint sets of variables ${}_c\mathcal{V} = ({}_c\mathcal{V}_i)_{i\in N}$ with ${}_c\mathcal{V}_i$ denoting a set of variables for interface identifier $i \in N$).

Definition 21 (Configuration term) The set of all *configuration terms* of sort $s \in S$ over a signature $\Sigma = (S, F, B)$, interface specification $S_i \in \mathcal{S}_i(S_p)$ over port specification $S_p \in \mathcal{S}_p(\Sigma)$, datatype variables ${}_d\mathcal{V}$, and component variables ${}_c\mathcal{V}$ is the smallest set ${}^s_c\mathcal{T}^{{}_c\mathcal{V}}_{{}_d\mathcal{V}}(\Sigma, S_i)$ satisfying the equations of Fig. 20. The set of all configuration terms of all sorts is denoted by ${}_c\mathcal{T}^{{}_c\mathcal{V}}_{{}_d\mathcal{V}}(\Sigma, S_i)$.

Note the use of function symbols of the corresponding datatype specification to build configuration terms. This allows one to reuse these function symbols when specifying architecture configurations. Moreover, note the use of port identifiers in the specification of configuration terms to denote the valuation of the corresponding port in an architecture configuration. For example, with *c.p* we denote the current valuation of port p of a component c.

The semantics of configuration terms is defined over an algebra and corresponding datatype variable assignments, an interface specification interpretation and corresponding *component variable assignments* (a family of mappings $\iota' = (\iota'_i)_{i\in N}$ with $\iota'_i\colon {}_c\mathcal{V}_i \to \mathsf{id}(cmp_i(J))$ for each interface identifier $i \in N$, where J is the corre-

$$
\begin{aligned}
v \in {}_d\mathcal{V}_s &\implies v \in {}^s_c\mathcal{T}^{{}_c\mathcal{V}}_{{}_d\mathcal{V}}(\Sigma, S_i)\ , \\
f \in F^0 &\implies f \in {}^s_c\mathcal{T}^{{}_c\mathcal{V}}_{{}_d\mathcal{V}}(\Sigma, S_i) \text{ [for } sort_0(f) = s]\ , \\
f \in F^{n+1} \wedge t_1 \in {}^{s_1}_c\mathcal{T}^{{}_c\mathcal{V}}_{{}_d\mathcal{V}}(\Sigma, S_i), \cdots, t_{n+1} \in {}^{s_{n+1}}_c\mathcal{T}^{{}_c\mathcal{V}}_{{}_d\mathcal{V}}(\Sigma, S_i) &\implies f(t_1, \cdots, t_{n+1}) \in {}^s_c\mathcal{T}^{{}_c\mathcal{V}}_{{}_d\mathcal{V}}(\Sigma, S_i) \text{ [for } n \in \mathbb{N},\ sort_0(f) = s, \\
&\qquad \text{and } sort_1(f) = s_1, \cdots, sort_{n+1}(f) = s_{n+1}]\ , \\
v \in {}_c\mathcal{V}_i \wedge p \in [Q_i]^2 \cup [Q_i]^3 &\implies v.p \in {}^s_c\mathcal{T}^{{}_c\mathcal{V}}_{{}_d\mathcal{V}}(\Sigma, S_i) \text{ [for } i \in N \text{ and } t^p(p) = s]\ .
\end{aligned}
$$

Figure 20 Inductive definition of configuration terms ${}^s_c\mathcal{T}^{{}_c\mathcal{V}}_{{}_d\mathcal{V}}(\Sigma, S_i)$ of sort $s \in S$ over signature $\Sigma = (S, F, B)$, interface specification $S_i = (N, Q)$, datatype variables ${}_d\mathcal{V}$, and component variables ${}_c\mathcal{V} = ({}_c\mathcal{V}_i)_{i \in N}$.

sponding interface specification interpretation), and an architecture configuration. In the following we denote with $\mathcal{I}'^{{}_c\mathcal{V}}_J$ the set of all component variable assignments for component variables and interface specification interpretation J.

Definition 22 (Configuration semantic function) The *configuration semantic function* for *configuration terms* ${}^s_c\mathcal{T}^{{}_c\mathcal{V}}_{{}_d\mathcal{V}}(\Sigma, S_i)$ over algebra $A \in \mathcal{A}(\Sigma)$ with corresponding datatype variable assignment $\iota \in \mathcal{I}^{{}_d\mathcal{V}}_A$, interface specification interpretation $J \in \mathcal{J}(S_i, A)$ with corresponding component variable assignment $\iota' \in \mathcal{I}'^{{}_c\mathcal{V}}_J$, and architecture configuration $k \in \mathcal{K}(cmp(J))$ is the function $[\![_]\!]^{(J,\iota')}_{(A,\iota)}(k)\colon {}^s_c\mathcal{T}^{{}_c\mathcal{V}}_{{}_d\mathcal{V}}(\Sigma, S_i) \to \alpha(s)$ characterized by the equations in Fig. 21.

Again, the semantics of a term is given by a function assigning a value of a corresponding set of the underlying algebra to each term.

5.1.2 Configuration Assertion

Assertions for architecture configurations are built over the corresponding configuration terms by the common logical operators and some dedicated predicates for the specification of so-called activation and connection constraints.

Definition 23 (Configuration assertion) The set of all *configuration assertions* over a signature Σ, interface specification $S_i \in \mathcal{S}_i(S_p)$ over port specification $S_p \in \mathcal{S}_p(\Sigma)$, datatype variables ${}_d\mathcal{V}$, and component variables ${}_c\mathcal{V}$ is the smallest set ${}_c\Gamma^{{}_c\mathcal{V}}_{{}_d\mathcal{V}}(\Sigma, S_i)$ satisfying the equations in Fig. 22.

Note that the following predicates can be used for the specification of configuration assertions:

- *Activation contstraints* can be used to denote activation and deactivation of components. With $\|c\|$, e.g., we denote the activation of component c. With $|i|_n$, $|i|^n$ we denote the constraint that at least n or at most n components with interface i are active at each point in time.
- *Connection constraints* can be used to denote constraints over the connection between components. With $c.p \to c'.p'$, e.g., we denote a constraint that port p of component c has to be connected to port p' of component c'. Finally, with $i.p \to j.p'$ we denote that port p of every component with interface i is connected to port p' of every component with interface j.

The semantics of configuration assertions is defined over an algebra with corresponding datatype variable assignment and an interface interpretation with corresponding component variable assignment.

Definition 24 (Configuration models relation) The *configuration models relation* for *configuration assertions* ${}_c\Gamma^{{}_c\mathcal{V}}_{{}_d\mathcal{V}}(\Sigma, S_i)$ over algebra $A \in \mathcal{A}(\Sigma)$ with corresponding datatype variable assignment $\iota \in \mathcal{I}^{{}_d\mathcal{V}}_A$, interface interpretation $J \in \mathcal{J}(S_i, A)$ with corresponding component variable assignments $\iota' \in \mathcal{I}'^{{}_c\mathcal{V}}_J$ is the relation $\iota, \iota', _ \models^J_A _ \subseteq \mathcal{K}(cmp(J)) \times {}_c\Gamma^{{}_c\mathcal{V}}_{{}_d\mathcal{V}}(\Sigma, S_i)$ characterized by the equations in Fig. 23. A configuration assertion γ is *valid* for architecture configuration $k \in \mathcal{K}(cmp(J))$ under algebra $A \in \mathcal{A}(\Sigma)$ and interface interpretation $J \in \mathcal{J}(S_i, A)$ iff there exists corresponding datatype variable assignment $\iota \in \mathcal{I}^{{}_d\mathcal{V}}_A$ and component variable assignment $\iota' \in \mathcal{I}'^{{}_c\mathcal{V}}_J$, such that $\iota, \iota', k \models^J_A \gamma$. Configuration k is a *model* of γ, written $k \models^J_A \gamma$ iff for each corresponding datatype variable assignment ι and component variable assignment ι' we have $\iota, \iota', k \models^J_A \gamma$. Configuration k is a *model* of a set of configuration assertions Γ, written $k \models^J_A \Gamma$ iff $k \models^J_A \gamma$ for each $\gamma \in \Gamma$.

Note that the semantics of configuration assertions is given in terms of a relation, parametrized by an interface interpretation. It determines all valid architecture configurations (over the components provided by the interface interpretation) for a given configuration assertion.

$$
\begin{aligned}
[\![v]\!]^{(J,\iota')}_{(A,\iota)}(k) &\overset{\text{def}}{=} \iota_s(v) \text{ [for } v \in {}_d\mathcal{V}_s] \ , \\
[\![f]\!]^{(J,\iota')}_{(A,\iota)}(k) &\overset{\text{def}}{=} \beta(f) \text{ [for function symbol } f \in F^0] \ , \\
[\![f(t_1, \cdots, t_n)]\!]^{(J,\iota')}_{(A,\iota)}(k) &\overset{\text{def}}{=} \beta(f)([\![t_1]\!]^{\iota}_{A}, \cdots, [\![t_n]\!]^{\iota}_{A}) \text{ [for function symbol } f \in F^{n+1}] \ , \\
[\![v.p]\!]^{(J,\iota')}_{(A,\iota)}(k) &\overset{\text{def}}{=} val_k(\iota'_i(v))(p) \text{ [for } i \in N \text{ and } v \in {}_c\mathcal{V}_i] \ .
\end{aligned}
$$

Figure 21 Recursive definition of semantic function for configuration terms ${}^{s}_{c}\mathcal{T}^{{}_c\mathcal{V}}_{{}_d\mathcal{V}}(\Sigma, S_i)$ with signature Σ, interface specification $S_i = (N, Q) \in \mathcal{S}_i(S_p)$ over port specification $S_p \in \mathcal{S}_p(\Sigma)$, datatype variables ${}_d\mathcal{V}$, component variables ${}_c\mathcal{V} = ({}_c\mathcal{V}_i)_{i \in N}$, algebra $A \in \mathcal{A}(\Sigma)$ and corresponding datatype variable assignment $\iota \in \mathcal{I}^{{}_d\mathcal{V}}_{A}$, interface specification interpretation $J \in \mathcal{J}(S_i, A)$ and corresponding component variable assignment $\iota' = (\iota'_i)_{i \in N} \in \mathcal{I}'^{{}_c\mathcal{V}}_{J}$, and architecture configuration $k = (C', N) \in \mathcal{K}(cmp(J))$.

$$
\begin{aligned}
b \in B^0 &\implies b \in {}_c\Gamma^{{}_c\mathcal{V}}_{{}_d\mathcal{V}}(\Sigma, S_i) \ , \\
b \in B^{n+1} \wedge t_1 \in {}^{s_1}_{c}\mathcal{T}^{{}_c\mathcal{V}}_{{}_d\mathcal{V}}(\Sigma, S_i), \cdots, t_{n+1} \in {}^{s_{n+1}}_{c}\mathcal{T}^{{}_c\mathcal{V}}_{{}_d\mathcal{V}}(\Sigma, S_i) &\implies b(t_1, \cdots, t_{n+1}) \in {}_c\Gamma^{{}_c\mathcal{V}}_{{}_d\mathcal{V}}(\Sigma, S_i) \\
&\qquad \text{[for } n \in \mathbb{N} \text{ and } sort_1(b) = s_1, \cdots, sort_{n+1}(b) = s_{n+1}] \ , \\
t, t' \in {}_c\mathcal{T}^{{}_c\mathcal{V}}_{{}_d\mathcal{V}}(\Sigma, S_i) &\implies t = t' \in {}_c\Gamma^{{}_c\mathcal{V}}_{{}_d\mathcal{V}}(\Sigma, S_i) \ , \\
E \in {}_c\Gamma^{{}_c\mathcal{V}}_{{}_d\mathcal{V}}(\Sigma, S_i) &\implies \neg E \in {}_c\Gamma^{{}_c\mathcal{V}}_{{}_d\mathcal{V}}(\Sigma, S_i) \ , \\
E, E' \in {}_c\Gamma^{{}_c\mathcal{V}}_{{}_d\mathcal{V}}(\Sigma, S_i) &\implies E \wedge E',\ E \vee E',\ E \implies E',\ E \iff E' \in {}_c\Gamma^{{}_c\mathcal{V}}_{{}_d\mathcal{V}}(\Sigma, S_i) \ , \\
E \in {}_c\Gamma^{{}_c\mathcal{V}}_{{}_d\mathcal{V}}(\Sigma, S_i) \wedge x \in {}_d\mathcal{V}_s &\implies \forall x.\ E \in {}_c\Gamma^{{}_c\mathcal{V}}_{{}_d\mathcal{V}}(\Sigma, S_i) \ \wedge \\
&\qquad \exists x.\ E \in {}_c\Gamma^{{}_c\mathcal{V}}_{{}_d\mathcal{V}}(\Sigma, S_i) \text{ [for } s \in S] \ , \\
E \in {}_c\Gamma^{{}_c\mathcal{V}}_{{}_d\mathcal{V}}(\Sigma, S_i) \wedge x \in {}_c\mathcal{V}_i &\implies \forall x.\ E \in {}_c\Gamma^{{}_c\mathcal{V}}_{{}_d\mathcal{V}}(\Sigma, S_i) \ \wedge \\
&\qquad \exists x.\ E \in {}_c\Gamma^{{}_c\mathcal{V}}_{{}_d\mathcal{V}}(\Sigma, S_i) \text{ [for } i \in N] \ , \\
n, m \in \mathbb{N} \wedge n \leq m &\implies |i|_n \in {}_c\Gamma^{{}_c\mathcal{V}}_{{}_d\mathcal{V}}(\Sigma, S_i) \wedge |i|^m \in {}_c\Gamma^{{}_c\mathcal{V}}_{{}_d\mathcal{V}}(\Sigma, S_i) \ \wedge \\
&\qquad |i|^m_n \in {}_c\Gamma^{{}_c\mathcal{V}}_{{}_d\mathcal{V}}(\Sigma, S_i) \text{ [for some } i \in N] \ , \\
v \in {}_c\mathcal{V}_i &\implies \|v\| \in {}_c\Gamma^{{}_c\mathcal{V}}_{{}_d\mathcal{V}}(\Sigma, S_i) \text{ [for some } i \in N] \ , \\
v \in {}_c\mathcal{V}_i \wedge v' \in {}_c\mathcal{V}_j \wedge p \in [Q_i]^2 \wedge p' \in [Q_j]^3 &\implies v.p \rightarrow v'.p' \in {}_c\Gamma^{{}_c\mathcal{V}}_{{}_d\mathcal{V}}(\Sigma, S_i) \text{ [for some } i, j \in N] \ , \\
p \in [Q_i]^2 \wedge p' \in [Q_j]^3 &\implies i.p \rightarrow j.p' \in {}_c\Gamma^{{}_c\mathcal{V}}_{{}_d\mathcal{V}}(\Sigma, S_i) \text{ [for some } i, j \in N] \ .
\end{aligned}
$$

Figure 22 Inductive definition of configuration assertions ${}_c\Gamma^{{}_c\mathcal{V}}_{{}_d\mathcal{V}}(\Sigma, S_i)$ over signature Σ, interface specification $S_i = (N, Q)$, datatype variables ${}_d\mathcal{V}$, and component variables ${}_c\mathcal{V} = ({}_c\mathcal{V}_i)_{i \in N}$.

5.1.3 Configuration Trace Assertion

Configuration trace assertions are a means to directly specify sets of configuration traces. They are formulated as temporal logic formulas over configuration assertions and consist of the common temporal operators and so-called rigid variables for datatypes and components. Compared to "normal" variables, which are newly interpreted at each point in time, these variables are interpreted only once for the whole execution trace.

Thus, we assume the existence of a set of *rigid datatype variables* (a family of disjoint sets of variables ${}^{r}_{d}\mathcal{V} = ({}^{r}_{d}\mathcal{V}_s)_{s \in S}$ with ${}^{r}_{d}\mathcal{V}_s$ denoting a set of variables for each sort $s \in S$) and *rigid component variables* (a family of disjoint sets of variables ${}^{r}_{c}\mathcal{V} = ({}^{r}_{c}\mathcal{V}_i)_{i \in N}$ with ${}^{r}_{c}\mathcal{V}_i$ denoting a set of variables for each interface identifier $i \in N$).

Definition 25 (Configuration trace assertion) The set of all configuration trace assertions over signature Σ, interface specification $S_i \in \mathcal{S}_i(S_p)$ over port specification S_p, datatype variables ${}_d\mathcal{V}$, rigid datatype variables ${}^{r}_{d}\mathcal{V}$, component variables ${}_c\mathcal{V}$, and rigid component variables ${}^{r}_{c}\mathcal{V}$ is the smallest set ${}_t\Gamma^{({}_d\mathcal{V}, {}_c\mathcal{V})}_{({}^{r}_{d}\mathcal{V}, {}^{r}_{c}\mathcal{V})}(\Sigma, S_i)$ satisfying the equations in Fig. 24.

$$
\begin{aligned}
\iota,\iota',k \models^J_A b &\iff \gamma(b) \text{ [for } b\in B^0] \ ,\\
\iota,\iota',k \models^J_A b(t_1,\cdots,t_n) &\iff \gamma(b)([\![t_1]\!]^\iota_A,\cdots,[\![t_n]\!]^\iota_A) \text{ [for } b\in B^{n+1}] \ ,\\
\iota,\iota',k \models^J_A t=t' &\iff [\![t]\!]^\iota_A = [\![t']\!]^\iota_A \ ,\\
\iota,\iota',k \models^J_A E\wedge E' &\iff \iota,\iota',k \models^J_A E \wedge \iota,\iota',k \models^J_A E' \ ,\\
\iota,\iota',k \models^J_A E\vee E' &\iff \iota,\iota',k \models^J_A E \vee \iota,\iota',k \models^J_A E' \ ,\\
\iota,\iota',k \models^J_A E \implies E' &\iff \iota,\iota',k \models^J_A E \implies \iota,\iota',k \models^J_A E' \ ,\\
\iota,\iota',k \models^J_A E \iff E' &\iff \iota,\iota',k \models^J_A E \iff \iota,\iota',k \models^J_A E' \ ,\\
\iota,\iota',k \models^J_A \exists x.\ E &\iff \exists x'\in\alpha(s)\colon \iota[s\colon x\mapsto x'],\iota',k \models^J_A E \text{ [for } s\in S \text{ and } x\in {}_d\mathcal{V}_s] \ ,\\
\iota,\iota',k \models^J_A \forall x.\ E &\iff \forall x'\in\alpha(s)\colon \iota[s\colon x\mapsto x'],\iota',k \models^J_A E \text{ [for } s\in S \text{ and } x\in {}_d\mathcal{V}_s] \ ,\\
\iota,\iota',k \models^J_A \exists x.\ E &\iff \exists x'\in \mathsf{id}(cmp_i(J))\colon \iota,\iota'[i\colon x\mapsto x'],k \models^J_A E \text{ [for } i\in N \text{ and } x\in {}_c\mathcal{V}_i] \ ,\\
\iota,\iota',k \models^J_A \forall x.\ E &\iff \forall x'\in \mathsf{id}(cmp_i(J))\colon \iota,\iota'[i\colon x\mapsto x'],k \models^J_A E \text{ [for } i\in N \text{ and } x\in {}_c\mathcal{V}_i] \ ,\\
\iota,\iota',k \models^J_A |i|_n &\iff |cmp_i(J)\cap C'| \geq n \text{ [for } k=(C',N) \text{ and } i\in N] \ ,\\
\iota,\iota',k \models^J_A |i|^m &\iff |cmp_i(J)\cap C'| \leq m \text{ [for } k=(C',N) \text{ and } i\in N] \ ,\\
\iota,\iota',k \models^J_A |i|^m_n &\iff \iota,\iota',k \models^J_A |i|_n \ \wedge\ \iota,\iota',k \models^J_A |i|^m \ ,\\
\iota,\iota',k \models^J_A \|v\| &\iff \iota'_i(v)\in \mathsf{id}(C') \text{ [for } k=(C',N),\ i\in N, \text{ and } v\in {}_c\mathcal{V}_i] \ ,\\
\iota,\iota',k \models^J_A v.p\to v'.p' &\iff (\iota'(v'),p')\in N(\iota'(v),p) \text{ [for } k=(C',N)] \ ,\\
\iota,\iota',k \models^J_A i.p\to j.p' &\iff \iota,\iota',k \models^J_A \forall v,v'.\ \big(\|v\|\wedge\|v'\| \implies v.p\to v'.p'\big) \text{ [for } v\in {}_c\mathcal{V}_i,\ v'\in {}_c\mathcal{V}_j] \ .
\end{aligned}
$$

Figure 23 Recursive definition of models relation for configuration assertions ${}_c\Gamma^{S_i}_\Sigma({}_d\mathcal{V},{}_c\mathcal{V})$ with interface specification $S_i=(N,Q)$, algebra $A\in\mathcal{A}(\Sigma)$ with corresponding datatype variable assignments $\iota=(\iota_s)_{s\in S}$, and interface specification interpretation $J =\in \mathcal{J}(S_i,A)$ with corresponding component variable assignment $\iota'=(\iota'_i)_{i\in N}$.

$$
\begin{aligned}
\phi \in {}_c\Gamma^{{}_c\mathcal{V}\cup{}^r_c\mathcal{V}}_{{}_d\mathcal{V}\cup{}^r_d\mathcal{V}}(\Sigma,S_i) &\implies \phi\in {}_t\Gamma^{({}_d\mathcal{V},{}_c\mathcal{V})}_{({}^r_d\mathcal{V},{}^r_c\mathcal{V})}(\Sigma,S_i) \ ,\\
\gamma \in {}_t\Gamma^{({}_d\mathcal{V},{}_c\mathcal{V})}_{({}^r_d\mathcal{V},{}^r_c\mathcal{V})}(\Sigma,S_i) &\implies \bigcirc\gamma,\ \Diamond\gamma,\ \Box\gamma \in {}_t\Gamma^{({}_d\mathcal{V},{}_c\mathcal{V})}_{({}^r_d\mathcal{V},{}^r_c\mathcal{V})}(\Sigma,S_i) \ ,\\
\gamma,\gamma' \in {}_t\Gamma^{({}_d\mathcal{V},{}_c\mathcal{V})}_{({}^r_d\mathcal{V},{}^r_c\mathcal{V})}(\Sigma,S_i) &\implies (\gamma\ \mathcal{U}\ \gamma'),\ (\gamma\ \mathcal{W}\ \gamma') \in {}_t\Gamma^{({}_d\mathcal{V},{}_c\mathcal{V})}_{({}^r_d\mathcal{V},{}^r_c\mathcal{V})}(\Sigma,S_i) \ ,\\
x\in {}^r_d\mathcal{V}_s \wedge \gamma \in {}_t\Gamma^{({}_d\mathcal{V},{}_c\mathcal{V})}_{({}^r_d\mathcal{V},{}^r_c\mathcal{V})}(\Sigma,S_i) &\implies \forall x.\ \gamma\in {}_t\Gamma^{({}_d\mathcal{V},{}_c\mathcal{V})}_{({}^r_d\mathcal{V},{}^r_c\mathcal{V})}(\Sigma,S_i)\ \wedge\\
&\qquad \exists x.\ \gamma\in {}_t\Gamma^{({}_d\mathcal{V},{}_c\mathcal{V})}_{({}^r_d\mathcal{V},{}^r_c\mathcal{V})}(\Sigma,S_i) \text{ [for } s\in S] \ ,\\
x\in {}^r_c\mathcal{V}_i \wedge \gamma \in {}_t\Gamma^{({}_d\mathcal{V},{}_c\mathcal{V})}_{({}^r_d\mathcal{V},{}^r_c\mathcal{V})}(\Sigma,S_i) &\implies \forall x.\ \gamma\in {}_t\Gamma^{({}_d\mathcal{V},{}_c\mathcal{V})}_{({}^r_d\mathcal{V},{}^r_c\mathcal{V})}(\Sigma,S_i)\ \wedge\\
&\qquad \exists x.\ \gamma\in {}_t\Gamma^{({}_d\mathcal{V},{}_c\mathcal{V})}_{({}^r_d\mathcal{V},{}^r_c\mathcal{V})}(\Sigma,S_i) \text{ [for } i\in N] \ .
\end{aligned}
$$

Figure 24 Inductive definition of configuration trace assertions ${}_t\Gamma^{({}_d\mathcal{V},{}_c\mathcal{V})}_{({}^r_d\mathcal{V},{}^r_c\mathcal{V})}(\Sigma,S_i)$ over signature $\Sigma=(S,F,B)$, interface specification $S_i=(N,Q)$, datatype variables ${}_d\mathcal{V}=({}_d\mathcal{V}_s)_{s\in S}$, component variables ${}_c\mathcal{V}=({}_c\mathcal{V}_i)_{i\in N}$, rigid datatype variables ${}^r_d\mathcal{V}=({}^r_d\mathcal{V}_s)_{s\in S}$, and rigid component variables ${}^r_c\mathcal{V}=({}^r_c\mathcal{V}_i)_{i\in N}$.

The semantics of configuration trace assertions is given according to [24]. It is defined over an algebra, an interface interpretation, *rigid datatype variable assignments* (a family of mappings $\kappa=(\kappa_s)_{s\in S}$ with $\kappa_s\colon {}_d\mathcal{V}_s\to\alpha(s)$ for each sort $s\in S$) and *rigid component variable assignments* (a family of mappings $\kappa'=(\kappa'_i)_{i\in N}$ with $\kappa'_i\colon {}^r_c\mathcal{V}_i\to\mathsf{id}(cmp_i(J))$ for each interface identifier $i\in N$, where J is the corresponding interface specification interpretation). It relates each configuration trace assertion with a configuration trace and point in time such that the trace satisfies the assertion at the corresponding time point.

Definition 26 (Configuration trace models relation) The *configuration trace models relation* for *configuration trace assertions* ${}_t\Gamma^{({}_d\mathcal{V},{}_c\mathcal{V})}_{({}^r_d\mathcal{V},{}^r_c\mathcal{V})}(\Sigma,S_i)$ over algebra $A\in\mathcal{A}(\Sigma)$ with corresponding datatype variable assignment $\iota\in\mathcal{I}^{{}_d\mathcal{V}}_A$ and rigid datatype variable assignment $\kappa\in\mathcal{K}^{{}_d\mathcal{V}}_A$, interface interpretation $J\in\mathcal{J}(S_i,A)$, with corresponding component variable assignment $\iota'\in{\mathcal{I}'}^{{}_c\mathcal{V}}_J$ and rigid component variable assignment $\kappa'\in$

$$
\begin{aligned}
\kappa, \kappa', (t,n) \models^J_A \phi &\iff \exists \iota \in \mathcal{I}^{_d\mathcal{V}}_A, \iota' \in \mathcal{I'}^{_c\mathcal{V}}_J \colon \iota \cup \kappa, \iota' \cup \kappa', t(n) \models^J_A \phi \ , \\
\kappa, \kappa', (t,n) \models^J_A \bigcirc\gamma &\iff \kappa, \kappa', (t, n+1) \models^J_A \gamma \ , \\
\kappa, \kappa', (t,n) \models^J_A \Diamond\gamma &\iff \exists n' \geq n \colon \kappa, \kappa', (t, n') \models^J_A \gamma \ , \\
\kappa, \kappa', (t,n) \models^J_A \Box\gamma &\iff \forall n' \geq n \colon \kappa, \kappa', (t, n') \models^J_A \gamma \ , \\
\kappa, \kappa', (t,n) \models^J_A \left(\gamma \,\mathcal{U}\, \gamma'\right) &\iff \exists n' \geq n \colon \kappa, \kappa', (t, n') \models^J_A \gamma' \ \wedge \ \forall n \leq m < n' \colon \kappa, \kappa', (t, m) \models^J_A \gamma \ , \\
\kappa, \kappa', (t,n) \models^J_A \left(\gamma \,\mathcal{W}\, \gamma'\right) &\iff \kappa, \kappa', (t,n) \models^J_A \left(\gamma \,\mathcal{U}\, \gamma'\right) \ \vee \ \kappa, \kappa', (t,n) \models^J_A \Box\gamma \ , \\
\kappa, \kappa', (t,n) \models^J_A \exists x.\ \gamma &\iff \exists x' \in \alpha(X) \colon \kappa[X \colon x \mapsto x'], \kappa', (t,n) \models^J_A \gamma \text{ [for } X \in S] \ , \\
\kappa, \kappa', (t,n) \models^J_A \forall x.\ \gamma &\iff \forall x' \in \alpha(X) \colon \kappa[X \colon x \mapsto x'], \kappa', (t,n) \models^J_A \gamma \text{ [for } X \in S] \ , \\
\kappa, \kappa', (t,n) \models^J_A \exists x.\ \gamma &\iff \exists x' \in J^{-1}(X) \colon \kappa, \kappa'[X \colon x \mapsto x'], (t,n) \models^J_A \gamma \text{ [for } X \in N] \ , \\
\kappa, \kappa', (t,n) \models^J_A \forall x.\ \gamma &\iff \forall x' \in J^{-1}(X) \colon \kappa, \kappa'[X \colon x \mapsto x'], (t,n) \models^J_A \gamma \text{ [for } X \in N] \ .
\end{aligned}
$$

Figure 25 Recursive definition of models relation for configuration trace assertions ${}_t\Gamma^{({}_d\mathcal{V},{}_c\mathcal{V})}_{({}^r_d\mathcal{V},{}^r_c\mathcal{V})}(\Sigma, S_i)$ over signature $\Sigma = (S, F, B)$, interface specification $S_i = (N, Q)$, algebra $A = (S', F', B', \alpha, \beta, \gamma)$ with corresponding rigid datatype variable assignment $\kappa = (\kappa_s)_{s \in S}$, and interface specification interpretation $J = J \in \mathcal{J}(S_i, Q)$ with corresponding rigid component variable assignment $\kappa' = (\kappa'_i)_{i \in N}$.

$\mathcal{K'}^{_c\mathcal{V}}_J$ is the relation $\kappa, \kappa', _ \models^J_A _ \subseteq (\mathcal{K}(cmp(J)) \times \mathbb{N}) \times {}_t\Gamma^{({}_d\mathcal{V},{}_c\mathcal{V})}_{({}^r_d\mathcal{V},{}^r_c\mathcal{V})}(\Sigma, S_i)$ characterized by the equations in Fig. 25.

A configuration trace assertion γ is *valid* for configuration trace $t \in \mathcal{R}(cmp(J))$ under algebra $A \in \mathcal{A}(\Sigma)$ and interface interpretation $J = J \in \mathcal{J}(S_i, A)$ iff there exists corresponding rigid datatype variable assignment $\kappa \in \mathcal{K}^{_d\mathcal{V}}_A$ and rigid component variable assignment $\kappa' \in \mathcal{K'}^{_c\mathcal{V}}_J$, such that $\kappa, \kappa', (t, 0) \models^J_A \gamma$. Trace t is a *model* of γ, written $t \models^J_A \gamma$ iff for each corresponding rigid datatype variable assignment κ and rigid component variable assignment κ' we have $\kappa, \kappa', (t, 0) \models^J_A \gamma$. Trace t is a *model* of a set of configuration trace assertions Γ, written $t \models^J_A \Gamma$ iff $t \models^J_A \gamma$ for each $\gamma \in \Gamma$.

Note the existential quantification for datatype variable assignments and component variable assignments meaning that these variables are interpreted at each point in time, compared to the rigid once.

5.2 Specifying Configuration Trace Assertions

Configuration trace assertions can be specified by means of configuration trace specification templates (Fig. 26). Each template has a name and can import interface specification templates by means of their name. Then, a list of variables for the different sorts/interfaces are defined. Finally, a list of configuration trace assertions are formulated over the variables and interfaces specified by the corresponding interface specifications.

Spec Name	**uses** *ifSpec*
var var1, var2`Sort1`	
var3: `Sort2`	
assertion1 (var1, var2, var3)	
assertion2 (var1, var2, var3)	

Figure 26 Configuration trace specification template

5.3 Blackboard: Architecture Constraint Specification

In the following we provide an architecture constraint specification for Blackboard architectures. First, we specify constraints regarding the behavior of BBs and KSs, respectively. Then, we provide activation and connection constraints for such architectures by means of configuration trace specification templates.

5.3.1 Blackboard Behavior

A BB provides the *current state* towards solving the original problem and forwards problems and solutions from KSs. Fig. 27 provides a specification of the BBs behavior in terms of a configuration trace specification template consisting of three configuration trace assertions:

- if a solution to a subproblem is received on its input, then it is eventually provided at its output (Eq. 13).
- if solving a problem requires a set of subproblems to be solved first, those problems are eventually provided at its output (Eq. (14)).
- a problem is provided as long as it is not solved (Eq. (15)).

Spec Blackboard_Behavior	**uses** *Blackboard*
var bb:	BB
$p,\ p'$:	PROB
P:	PROB SET
s:	SOL

$$\square\Big((p,s) \in bb.n_s \implies \diamondsuit\big((p,s) \in bb.c_s\big)\Big) \tag{13}$$

$$\square\Big((p,P) \in bb.r_p \implies \big(\forall p' \in P.\ (\diamondsuit p' \in bb.o_p)\big)\Big) \tag{14}$$

$$\square\Big(p \in bb.o_p \implies \big(p \in bb.o_p\ \mathcal{W}\ (p, solve(p)) \in bb.n_s\big)\Big) \tag{15}$$

Figure 27 Architecture constraint specification: BB behavior.

5.3.2 Knowledge-Source Behavior

A KS receives open problems via o_p and solutions for other problems via c_s. It might contribute to the solution of the original problem by solving subproblems. Fig. 28 provides a specification of the KSs behavior in terms of a configuration trace specification template consisting of three configuration trace assertions:

- if a KS gets correct solutions for all the required subproblems, then it solves the problem eventually (Eq. (16)).
- in order to solve a problem, a KS requires solutions only for smaller problems (Eq. (17)).
- if a KS is able to solve a problem it will eventually communicate this (Eq. (18)).

5.3.3 Activation Constraints

Activation constraints for the Blackboard pattern are described by two configuration trace assertions provided in the configuration trace specification template in Fig. 29:

- Eq. (19) denotes the conditions that there is a unique BB component which is always activated (in contrast to KSs components which can be activated and deactivated arbitrarily).
- Eq. (20) requires that whenever a KS component offers to solve some problem, it is always activated when solutions to the required subproblems are provided.

5.3.4 Connection Constraints

Connection constraints are also specified by a configuration trace specification template provided in Fig. 30. It consists of two configuration trace assertions: Eq. (21) describes the required connections for all executions while Eq. (22) describes all connections which are not allowed. Roughly speaking the specification requires that for each point in time, input port o_p of a KS is connected (only) to output port o_p of the BB component, input port c_s of a KS is connected (only) to output port c_s of the BB component, output port r_p of a KS is connected (only) to input port r_p of the BB component, and output port n_s of a KS is connected (only) to input port n_s of the BB component.

5.4 Specifying Architecture Constraints

As stated in the introduction of this section, architecture constraints are specified in three steps by specifying datatypes, interfaces and configuration traces.

Definition 27 (Architecture constraint specification) An *architecture constraint specification* over datatype variables ${}_{d}\mathcal{V}$, ${}_{d}\mathcal{V}'$ and component variables ${}_{c}\mathcal{V}$, rigid datatype variables ${}^{r}_{d}\mathcal{V}$ and rigid component variables ${}^{r}_{c}\mathcal{V}$ is a 6-tuple $(\Sigma, \Phi, S_p, S_i, \Omega, \Gamma)$, consisting of:

- a signature Σ,
- a datatype specification $\Phi \subseteq {}_{d}\Gamma(\Sigma, {}_{d}\mathcal{V}')$,
- a port specification $S_p \in \mathcal{S}_p(\Sigma)$,
- an interface specifications $S_i = (N, Q) \in \mathcal{S}_i(S_p)$,
- a family of interface specifications $(\Omega_i)_{i \in N}$, where $\Omega_i \subseteq {}_{i}\Gamma_{{}_{d}\mathcal{V}}(\Sigma, Q_i)$ for each interface identifier $i \in N$, and
- a set of configuration trace assertions $\Gamma \subseteq {}_{t}\Gamma^{({}_{d}\mathcal{V}, {}_{c}\mathcal{V})}_{({}^{r}_{d}\mathcal{V}, {}^{r}_{c}\mathcal{V})}(\Sigma, S_i)$.

The semantics of an architecture constraint specification is given in terms of a set of configuration traces. Algorithm 1 describes how to systematically derive the semantics of an architecture constraint specification.

A set of configuration traces $T \subseteq \mathcal{R}(cmp(J))$ fulfills an architecture constraint specification $(\Sigma, \Phi, S_p, S_i, \Omega, \Gamma)$ over datatype variables ${}_{d}\mathcal{V}$, ${}_{d}\mathcal{V}'$ and component variables ${}_{c}\mathcal{V}$, rigid datatype variables ${}^{r}_{d}\mathcal{V}$ and rigid component variables iff there exists an algebra $A \subseteq \mathcal{A}(\Sigma)$ and interface specification interpretation J, such that:

Spec Knowledgesource_Behavior	**uses** *Blackboard*
var ks: KS	
$p,\ q$: `PROB`	
P: `PROB SET`	
$\Box\Big(\forall (p,P) \in ks.r_p.\ \big((\forall q \in P.\ \diamondsuit(q, solve(q)) \in ks.c_s) \implies \diamondsuit(p, solve(p)) \in ks.n_s\big)\Big)$	(16)
$\Box\Big(\forall (p,P) \in ks.r_p.\ \forall q \in P.\ q \prec p\Big)$	(17)
$\Box\Big(p \in ks.prob \wedge p \in ks.o_p \implies \diamondsuit(\exists P.\ (p,P) \in ks.r_p)\Big)$	(18)

Figure 28 Architecture constraint specification: KS behavior.

Spec Blackboard_Activation	**uses** *Blackboard*
var $bb,\ bb'$: BB	
$p,\ q$: `PROB`	
P: `PROB SET`	
$\Box(\|bb\| \wedge \forall bb'.\ bb' = bb)$	(19)
$\Box\Big(\forall (p,P) \in ks.r_p.\ \big(\forall q \in P.\ (\diamondsuit(q, solve(q)) \in bb.c_s) \implies \diamondsuit((q, solve(q)) \in bb.c_s \wedge \|ks\|)\big)\Big)$	(20)

Figure 29 Architecture constraint specification: activation specification.

Spec Connection	**uses** *Blackboard*
var bb: BB	
ks: KS	
$\Box\Big(ks.o_p \to bb.o_p \wedge ks.c_s \to bb.c_s \wedge bb.r_p \to ks.r_p \wedge bb.n_s \to ks.n_s\Big)$	(21)
$\Box\Big(\neg\big(ks.o_p \to bb.c_s\big) \wedge \neg\big(ks.c_s \to bb.o_p\big) \wedge \neg\big(bb.r_p \to ks.n_s\big) \wedge \neg\big(bb.n_s \to ks.r_p\big)\Big)$	(22)

Figure 30 architecture constraint specification: connection specification.

Algorithm 1 Calculate semantics of Architecture constraint specification

Input: architecture constraint specification S with:

- signature Σ,
- datatype specification $\Phi \subseteq {}_d\Gamma(\Sigma, {}_d\mathcal{V}')$,
- port specification $S_p \in \mathcal{S}_p(\Sigma)$,
- interface specifications $S_i = (N, Q) \in \mathcal{S}_i(S_p)$,
- family of interface specifications $(\Omega_i)_{i \in N}$, where $\Omega_i \subseteq {}_i\Gamma_{{}_d\mathcal{V}}(\Sigma, Q_i)$ for each interface identifier $i \in N$, and
- set of configuration trace assertions $\Gamma \subseteq {}_t\Gamma^{({}_d\mathcal{V}, {}_c\mathcal{V})}_{({}^r_d\mathcal{V}, {}^r_c\mathcal{V})}(\Sigma, S_i)$.

Output: a set of configuration traces T satisfying S

$A \Leftarrow \mathcal{A}(\Sigma)$, such that $A \models \Phi$
$J \Leftarrow \mathcal{J}(S_i, A)$, such that $\forall i \in N, j \in J_i : j \models_A \Omega_i$
$T \Leftarrow t \in \mathcal{R}(cmp(J))$, such that $t \models^J_A \Gamma$
return T

- A is a model of the datatype specification: $A \models \Phi$,
- J_i satisfies the corresponding interface assertion: $j \models_A \Omega_i$,
- each trace $t \in T$ is a model of Γ: $t \models^J_A \Gamma$.

Fig. 5.4 depicts the relationship of the syntactic and semantic concepts to specify architecture properties. First, a datatype specification determines an algebra with corresponding sets of messages and operations on those sets. Later on, an interface specification determines a set of components valuated by messages from the corresponding algebra. Finally, the set of configuration trace assertions determine a set of configuration trace over those components. Note that configuration trace assertions may use certain operations specified in the corresponding datatype specification which is why configuration traces depend on the concrete interpretation of those operations.

5.5 Summary

Table 4 provides an overview of the concepts introduced in this section. For each concept it provides a brief description and related notation.

6 Configuration Diagrams

Configuration trace assertions are actually sufficient to specify each property of dynamic architectures. How-

Table 4 Overview of concepts to specify architecture constraints.

Concept	*Description*	*Related Notation*
component variable	variable for component identifiers	${}_{c}\mathcal{V}$
configuration term	term over signature Σ, interface specification S_i, datatype variables ${}_{d}\mathcal{V}$, and component variables ${}_{c}\mathcal{V}$	${}_{c}\mathcal{T}^{{}_{c}\mathcal{V}}_{{}_{d}\mathcal{V}}(\Sigma, S_i)$
activation constraint	*constraint on activation and deactivation of components*	
connection constraint	*constraint on the connection between components*	
component variable assignment	assignment of component identifiers of interface specification interpretation J to component variables ${}_{c}\mathcal{V}$	$\mathcal{I}'^{{}_{c}\mathcal{V}}_{J}$
configuration semantic function	assigns elements of an algebra A to configuration terms under a certain datatype variable assignment ι, interface specification interpretation J and corresponding component variable assignment ι', and architecture configuration k	$[\![_]\!]^{(J,\iota')}_{(A,\iota)}(k)$
configuration assertion	formula over configuration terms with corresponding signature Σ, interface specification S_i, datatype variables ${}_{d}\mathcal{V}$, and component variables ${}_{c}\mathcal{V}$	${}_{c}\Gamma^{{}_{c}\mathcal{V}}_{{}_{d}\mathcal{V}}(\Sigma, S_i)$
configuration models relation	relates configuration assertions with architecture configurations under an algebra A and corresponding datatype variable assignment ι, and interface specification interpretation J and corresponding component variable assignment ι'	$\iota, \iota', _ \models^{J}_{A} _$
rigid datatype variable	datatype variable with fixed assignment for the whole execution	${}^{r}_{d}\mathcal{V}$
rigid component variable	component variable with fixed assignment for the whole execution	${}^{r}_{c}\mathcal{V}$
configuration trace assertion	formula over configuration assertions with corresponding signature Σ, interface specification S_i, datatype variables ${}_{d}\mathcal{V}$, component variables ${}_{c}\mathcal{V}$, rigid datatype variables ${}^{r}_{d}\mathcal{V}$, and rigid component variables ${}^{r}_{c}\mathcal{V}$	${}_{t}\Gamma^{({}_{d}\mathcal{V}, {}_{c}\mathcal{V})}_{({}^{r}_{d}\mathcal{V}, {}^{r}_{c}\mathcal{V})}(\Sigma, S_i)$
rigid datatype variable assignment	assignment of elements of an algebra A to a set of rigid datatype variables ${}_{d}\mathcal{V}$,	$\mathcal{K}^{{}^{r}_{d}\mathcal{V}}_{A}$
rigid component variable assignment	assignment of component identifiers of interface specification interpretation J to rigid component variables ${}_{c}\mathcal{V}$	$\mathcal{K}'^{{}^{r}_{c}\mathcal{V}}_{J}$
configuration trace models relation	relates configuration trace assertions with configuration traces under an algebra A and corresponding rigid datatype variable assignment κ, and interface specification interpretation J and corresponding rigid component variable assignment κ'	$\kappa, \kappa', _ \models^{J}_{A} _$
configuration trace specification template	structured technique to specify configuration traces	graphical
architecture constraint specification	a signature Σ, Φ, interface specification S_i, and a set of configuration trace assertions Γ	$(\Sigma, \Phi, S_p, S_i, \Omega, \Gamma)$

ever, sometimes, certain common constraints are better expressed by a so-called configuration diagram. Configuration diagrams complement configuration trace assertions and are well-suited to specify interfaces and introduce certain common connection and activation constraints in one graphical notation.

6.1 Simple Configuration Diagrams

In its simplest form, a configuration diagram is just a graphical representation of an interface specification. It consists of boxes and small circles, representing an interface identifier and corresponding local, input, and output ports. Thereby, transparent circles inside a component represent local ports, while white and black cir-

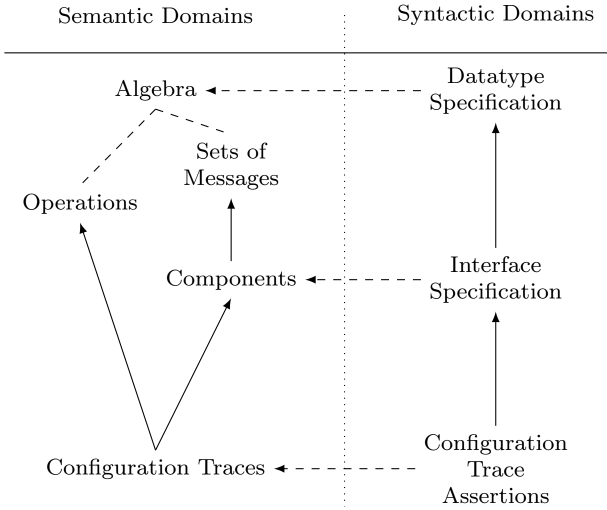

Figure 31 Relationship between syntactic and semantic concepts to specify architecture constraints.

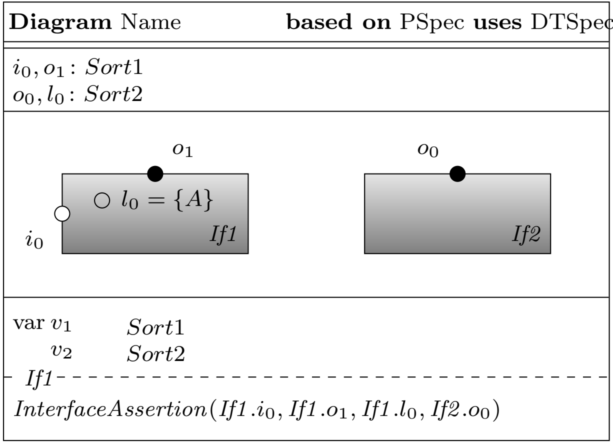

Figure 32 Simple configuration diagram as a graphical means to specify component types.

cles on the border of a component represent input and output ports, respectively. In addition, the ports are annotated with their name. The diagram is surrounded by a box which adds a name to the specification, a reference to imported port specifications, and a set of interface assertions.

Fig. 32 shows a graphical representation of a configuration diagram *Name* corresponding to an interface specification $S_i = (N, Q)$ and family of interface assertions Ω, with:

- interface identifiers $N = \{\mathit{If1}, \mathit{If2}\}$,
- interfaces $Q_{\mathit{If1}} = (\{i_0\}, \{o_1\})$ and $Q_{\mathit{If2}} = (\emptyset, \{o_0\})$, and
- interface assertions $\Omega_{\mathit{If1}} = \{\mathit{InterfaceAssertion}\}$.

As for interface specifications a configuration diagram may include also a corresponding port specification. Fig. 32, e.g., includes a port specification declaring ports

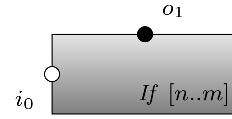

Figure 33 Min-max annotation requiring at least n but at most m active components of type *If* at each point in time.

l_0, i_0, o_0, o_1 and corresponding types $t^p(i_0) = t^p(o_1) = \mathit{Sort1}$ and $t^p(o_0) = t^p(l_0) = \mathit{Sort2}$.

6.2 Advanced Configuration Diagrams

Sometimes, it is convenient to annotate configuration diagrams by certain activation and connection constraints. These annotations are actually graphical synonyms for certain configuration trace assertions and they can be separated into activation and connection annotations.

6.3 Activation Annotations

Activation annotations enhance a configuration diagram by constraints regarding the activation and deactivation of certain components. Thus, they are modeled by predicates or mappings over interface identifiers.

6.3.1 Min-Max Annotations

Min-max annotations restrict the number of active components of a certain type.

Definition 28 (Min-max annotation) A *min-max annotation* for an interface specification (N, Q) is a pair of mappings (A_{min}, A_{max}), with A_{min}, $A_{max} : N \dashrightarrow \mathbb{N}$.

Note that not every interface needs to be annotated.

A min-max annotation can be easily specified by adding the two numbers to an interface identifier in a configuration diagram. Fig. 33, for example, depicts a constraint that at least n but at most m components of type *If* are active at each point in time.

The semantics of min-max annotations is given by means of corresponding configuration trace assertions.

Definition 29 (Min-max annotation semantics) The *semantics* of a *min-max annotation* (A_{min}, A_{max}), for interface specification (N, Q) is given by the following configuration trace assertion:

$$\Box\Big(\bigwedge_{i \in \mathrm{dom}(A_{min})} |i|_{A_{min}(i)} \wedge \bigwedge_{i \in \mathrm{dom}(A_{max})} |i|^{A_{max}(i)} \Big) \ . \quad (23)$$

Note that if $A_{min}(i) = A_{max}(i)$, then one number can be omitted and only one is to be annotated in the corresponding configuration diagram.

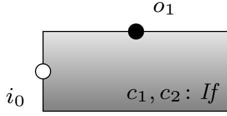

Figure 34 Rigid annotation requiring that only components c_1 and c_2 are activated throughout system execution.

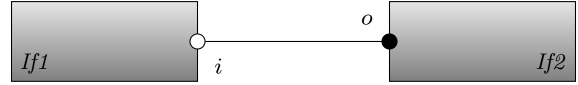

Figure 35 Required connection annotation requiring components of type *If1* to be always connected to a component of type *If2* through ports i and o, respectively.

6.3.2 Rigid Annotations

A min-max annotation does only constrain the number of components for a certain interface at a certain point in time. It does not say anything about which components these are. Assume, for example, we want to specify that a unique component of type i is active at each point in time. If we put a min-max constraint of 1 for interface i, than this means that exactly one component of type i is active at each point in time. However, it can be that at some point only component c_1 is active while at another time c_2 is active.

To specify that at each point in time the same components have to be activated we can use so-called rigid annotations.

Definition 30 (Rigid annotation) A *rigid annotation* for an interface specifications (N, Q) and rigid component variables ${}^r_c\mathcal{V} = ({}^r_c\mathcal{V}_i)_{i \in N}$ is a mapping $A_r\colon N \to \wp({}^r_c\mathcal{V})$, such that $\forall i \in N\colon A_r(i) \subseteq {}^r_c\mathcal{V}_i$.

A rigid annotation is specified by a list of variables for each interface. Note, however, that we require the use of *rigid* component variables here.

Fig. 34 depicts a constraint that only components c_1 and c_2 are activated throughout system execution.

The semantics of rigid annotations is given by means of configuration trace assertions.

Definition 31 (Rigid annotation semantics) The *semantics* of a *rigid annotation* A_r for interface specification (N, Q) is given by the following configuration trace assertion:

$$\Box\Big(\bigwedge_{i \in N} (\forall v\colon \bigvee_{c \in A_r(i)} (c = v))\Big) \ , \tag{24}$$

where $v \in {}_c\mathcal{V}$ is a (non-rigid) component variable.

6.4 Connection Annotations

Connection annotations enhance a configuration diagram by constraints regarding the connection of certain components. Thus, they are modeled by predicates or mappings over relations over interface ports.

Definition 32 (Required connection annotation) A *required connection annotation* for an interface specification S_i is a relation $A_c\colon \mathsf{in}(S_i) \dashrightarrow \mathsf{out}(S_i)$.

A required connection annotation is expressed by solid connections between the corresponding ports. Figure 35 denotes a constraint that a component of type *If1* is always connected to a component of type *If2* through ports i and o, respectively.

Definition 33 (Required connection annotation semantics) The *semantics* of a *required connection annotation* A_c for interface specification $S_i \in \mathcal{I}(S_p)$ over port specification $S_p \in \mathcal{S}_p(\Sigma)$ is given by the following configuration trace assertion:

$$\Box\Big(\bigwedge_{((j,i),(k,o)) \in A_c} j.i \to k.o \ \wedge \bigwedge_{((j,i),(k,o)) \in rest(A_c)} \neg\big(j.i \to k.o\big)\Big) \ , \tag{25}$$

where $rest(A_c) = \mathsf{in}(S_i) \times \mathsf{out}(S_i) \setminus A_c$.

Note that a required connection annotation induces a *full* homomorphism (a homomorphism preserving non-edge) between a configuration diagram and corresponding architecture configuration.

Property 3 (Required conn. induces full homomorphism) Let A_c be a required connection annotation for interface specification $S_i = (N, Q)$ with induced configuration trace assertion φ. Moreover, let $J \in \mathcal{J}(S_i, A)$ be a corresponding interface specification interpretation under an algebra $A = (S', F', B', \alpha, \beta, \gamma) \in \mathcal{A}(\Sigma)$. Finally, let $\delta\colon \mathsf{id}(cmp(J)) \to N$ denote the interface for each component identifier and $\delta^i = (\delta^i_d)_{d \in \mathsf{id}(cmp(J))}$ and $\delta^o = (\delta^o_d)_{d \in \mathsf{id}(cmp(J))}$ denoting the corresponding input and output port interpretations for component identifier c.

Then, (δ^i, δ^o) form a homomorphism from each architecture configuration of each trace $t \in \mathcal{R}(cmp(\delta))$ satisfying φ (for each point in time) to the corresponding possible connection annotation:

$$\begin{aligned} &\forall t \in \big\{t \in \mathcal{R}(cmp(J)) \mid t \models^J_A \varphi\big\}, n \in \mathbb{N}, c \in [t(n)]^1 , \\ &c' \in [t(n)]^1 , d = [c]^1 , d' = [c']^1 , p \in [c]^3 , p' \in [c]^4 : \\ &\quad ((d,p),(d',p')) \in [t(n)]^2 \iff \\ &\quad \Big((\delta(d), \delta^i_{\delta(d)}(p)), (\delta(d'), \delta^o_{\delta(d')}(p'))\Big) \in A_c \ . \end{aligned} \tag{26}$$

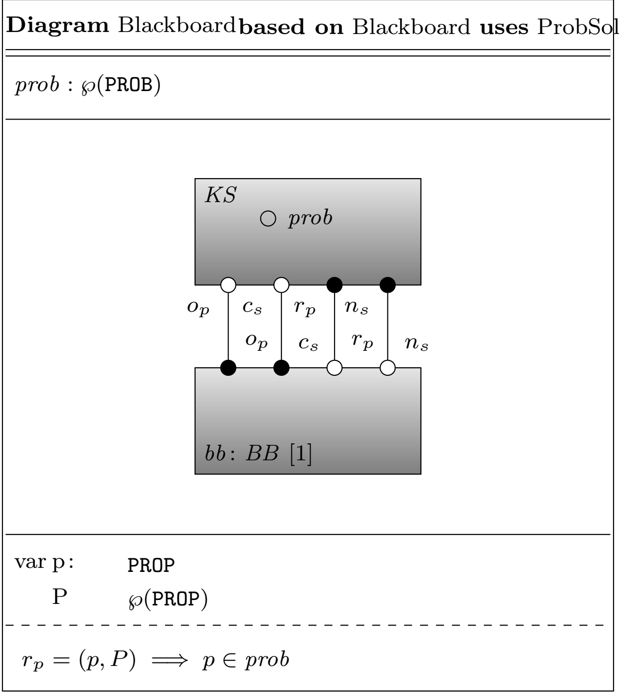

Figure 36 Configuration diagram for Blackboard architectures.

Proof Let $t \in \{t \in \mathcal{R}(cmp(J)) \mid t \models_A^J \varphi\}$, $n \in \mathbb{N}$, $c \in [t(n)]^1$, $c' \in [t(n)]^1$, $d = [c]^1$, $d' = [c']^1$, $p \in [c]^3$, and $p' \in [c]^4$.

Case 1 ($\implies$) Assume $((d,p),(d',p')) \in [t(n)]^2$ and show $\Big((\delta(d), \delta^i_{\delta(d)}(p)), (\delta(d'), \delta^o_{\delta(d')}(p'))\Big) \in A_c$ by contradiction. Thus, assume $\neg\Big((\delta(d), \delta^i_{\delta(d)}(p)), (\delta(d'), \delta^o_{\delta(d')}(p'))\Big) \in A_c$. The, from Def. 33, have that φ contains conjunction $\neg(j.i \to k.o)$. Thus, since $t \models_A^J \varphi$, have $((d,p),(d',p')) \notin [t(n)]^2$ by Def. 24 which is in contradiction with the assumption.

Case 2 ($\impliedby$) Assume $\Big((\delta(d), \delta^i_{\delta(d)}(p)), (\delta(d'), \delta^o_{\delta(d')}(p'))\Big) \in A_c$. Thus, from Def. 33, have that φ contains conjunction $j.i \to k.o$. Thus, since $t \models_A^J \varphi$, have $((d,p),(d',p')) \in [t(n)]^2$ by Def. 24.

6.5 Blackboard: Configuration Diagram

The interface specification of the Blackboard pattern as well as the activation and connection constraints of Eq. (19) and Eq. (21), respectively, could have been also expressed by the configuration diagram in Fig. 36:

- The interface specification is given by the two interfaces *KK* and *BB*, respectively.
- Eq. (19) is addressed by adding variable *bb* and the corresponding min-max annotation.
- Eq. (21) is addressed by the solid connections between the ports.

6.6 Refining Configuration Diagrams

Configuration diagrams are well-suited to specify interfaces and certain common activation and connection constraints.

However, not all constraints can be specified only by means of configuration diagrams which is why configuration diagrams are usually refined by adding further constraints by means of configuration trace assertions.

6.7 Summary

Table 5 provides an overview of the concepts introduced in this section. For each concept a brief description as well as related notation is provided.

7 Verifying a Specification

As demonstrated by the example, the approach allows for formal specification of patterns of dynamic architectures. Such a specification is useful, for example, to check pattern conformance of an architecture, i.e., whether a concrete architecture implements a certain pattern. On the other hand, having a formal specification of a pattern allows to formally analyze the specification.

This section demonstrates how a specification can be used to formally reason about it. Therefore, we specify a characteristic guarantee of Blackboard architectures by means of configuration trace assertions and prove it from the specification developed so far.

7.1 Specifying Properties

First, a property is specified over the architecture. The property can be formally specified by applying the techniques presented so far. As stated in the introduction of this article, one characteristic property of a Blackboard architecture is its ability to (collaboratively) solve a complex problem even if no single KS exists which is able to solve the problem on its own.

Theorem 1 *Assuming that KSs are active when required:*

$$\Box\Big(\forall p \in bb.o_p.\ \Diamond\big(\exists ks.\ p \in ks.prob\big)\Big)\ , \tag{27}$$

Table 5 Overview of concepts for configuration diagrams.

Concept	*Description*	*Related Notation*
configuration diagram	graphical notation to specify interfaces as well as common activation and connection constraints	graphical
activation annotations	annotations to constraint the activation/deactivation of components	
min-max annotation	*specifies the minimal(min)/maximal(max) number of components of a certain type*	$[min..max]$
rigid annotation	*set of rigid component variables denoting the set of all possible active components of an interface If*	$c_1, c_2 : If$
connection annotations	annotations to constrain the connection between components	
required connection annotation	*specification of required port-connection*	———

a Blackboard architecture guarantees to solve the original problem:

$$\Box\Big(p \in bb.r_p \implies \Diamond(p, solve(p)) \in bb.c_s\Big) \ . \tag{28}$$

7.2 Verifying The Specification

Then, the specification is verified w.r.t. the identified property by proving it from the specification. Therefore, the constraints introduced in the specification of the pattern serve as the major arguments throughout the proof. In the following we prove Thm. 1 by applying the different constraints specified for the pattern.

Proof The proof is by well-founded induction over the problem relation $\prec$: First, by Eq. (19) or Fig. 36 there exists a unique blackboard component bb which is always activated. Then, by Eq. (27), we are sure that for each problem eventually a `KnowledgeSource` ks exists which is capable to solve the problem. By Eq. (18) the ks will eventually communicate the subproblems p' it requires to solve the original problem p on its port r_p and this information is then transferred to port r_p of bb by the connection constraints imposed by Eq. (21) or Fig. 36. By Eq. (14), bb will provide these subproblems p' eventually on its output port o_p and publish it as long as it is not solved (as required by Eq. (15)). Since the subproblems p' provided to bb are strictly less than the original problem p (due to Eq. (17)), they will eventually be solved and its solutions s' provided on port c_s of bb by the induction hypothesis. ks will eventually be activated for each solution s' (Eq. (20)) and due to Eq. (21) or Fig. 36 the solutions s' are transferred to the corresponding port c_s of ks. Thus, ks eventually has all solutions s' to its subproblems p' and will then solve the original problem p by Eq. (16) and publish the solution s on its port n_s. Solution s is received eventually by bb on its port n_s due to Eq. (21) or Fig. 36 and is finally provided by bb on its port c_s due to Eq. (13). □

8 Discussion

The approach presented in this article is characterized by the following properties:

- *Formal*: The approach is based on a formal foundation with a formal semantics for each specification technique.
- *Uniform*: Each specification technique is based on a uniform model for dynamic architectures.
- *Abstract*: The approach is based on a rather abstract notion of architecture.
- *Model-theoretic semantics*: The semantics of each technique is given in terms of models which satisfy a corresponding specification.

These properties induce several benefits as well as some drawbacks which we will briefly discuss in the following.

8.1 Unambiguous Interpretation

Due to its formal nature, specifications can be interpreted as mathematical models. As demonstrated in Sect. 7, this enables formal analyses and verifications of the specifications.

8.2 Consistent Specification Techniques

Since the semantics of each technique is given in terms of a uniform model, inconsistencies can be detected more easily since the impact of different specification assertions can be directly related to each other.

8.3 Generality

Due to the abstract nature of the underlying model, the approach is very general. Its specifications and corresponding verification results can be interpreted for different, concrete architecture instances.

8.4 Extensibility

Since the semantics is given in terms of model-theory, this makes the approach easily extensible. New constructs can be easily integrated by describing its impact of the underlying model.

8.5 Stepwise Refinement

Another implication of the model-theoretic semantics is the possibility for stepwise refinement of specifications. After specifying a set of interfaces we can add more and more constraints. Thus, gradually lessen the space of possible architectures satisfying the specification.

8.6 Potential Limitations

Of course, the approach imposes some limitations which we will briefly discuss in the following.

8.6.1 Generality

While generality is listed as a benefit of the approach, it can also be seen as a drawback. Due to the abstract nature of the approach, it omits several details regarding component instantiation and communication between components. Thus, the approach is not well-suited in situations in which it is important to reason about these, more detailed aspects.

8.6.2 Consistency of Specifications

Although the approach is based on a uniform model of architectures, this does not ensure consistency of specifications developed with the approach. Constraints are expressed in temporal logic formulas which can be complex and induce inconsistencies of specifications.

8.6.3 Practical Evaluation

In this article we provided the theoretical foundations of our approach and demonstrated the concepts by means of a small example. To demonstrate the power of the approach it should be applied to more real world specifications.

9 Related Work

In this article we described an approach to specify constraints for dynamic architectures. Thus, related work can be found in three different areas. (i) specification of dynamic architectures, (ii) specification of architectural constraints, and (iii) specification of dynamic reconfiguration. In the following we briefly discuss each of them in more detail.

9.1 Specification of Dynamic Architectures

Over the last decades, several so-called Architecture Description Languages (ADLs) emerged to support in the formal specification of architectures. Some of them also support the specification of dynamic aspects [20, 22, 23, 3, 2, 29, 13, 16].

While ADLs support in the formal specification of (dynamic) architectures, they were developed with the aim to specify individual architecture instances, rather than architecture constraints which requires more abstract specification techniques.

By providing a language to specify such constraints, we actually complement these approaches. Our language can be used to specify architectural constraints and verify them against the concrete architectures specified in one of those languages.

9.2 Specification of Architectural Constraints

Attempts to formalize architectural styles and patterns required more abstract specification techniques and focused on the specification of architectural constraints, rather than concrete architectures.

Such constraints are either specified by a general specification language such as Z [32] (as, for example, by Abowd et al. [1]), algebraic specifications (as, for example, by Moriconi et al. [28] and Penix et al. [30]), graph grammars (as, for example, by Le MÃľtayer [21]) or by the use of process algebras (as, for example, by Bernardo et al. [5]) or directly from architectural primitives (as, for example, by Mehta and Medvidovic [27]).

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.

9.3 Specification of Dynamic Reconfigurations

Recently, some attempts were made to model dynamic reconfigurations in an abstract, language independent manner.

9.3.1 Stateless Reconfiguration

The first approaches in this area focused on plain structural evolution. Examples include the work of Le Métayer [21], Hirsch and Montanari [17], and Mavridou et al. [26]. While these approaches focus on the specification of constraints for dynamic architectures, similar as for ADLs, the relation of behavioral and structural aspects is not considered.

9.3.2 State-Full Reconfiguration

More recent approaches focus on the interrelation of behavior and configuration and are probably most closely related to our work.

One prominent example here is the work of Wermelinger et al. [35,34]. The authors introduce a graph based architecture reconfiguration language based on the unity language [12]. The authors recognize that the interplay between topology and run-time behavior is important and so their language also allows for the specification of reconfiguration constraints formulated over run-time behavior.

Bruni et al. [9] provide a graph-based approach to dynamic reconfiguration. Reconfigurations are modeled as typed graph grammars. Also here, the authors provide a mechanism to express architectural constraints.

Another approach in this area is the one of Batista et al. [4] where reconfiguration is specified as a set of reconfiguration rules. In a reconfiguration rule, a predicate is specified to trigger a reconfiguration and the result of a reconfiguration is specified in terms of attach and detach operations.

While these works actually recognize the relationship of behavior and state in the specification of dynamic reconfigurations, they usually focus on the specification of concrete architecture instances, rather than architecture constraints. The specification of constraints is only of secondary rule which is why they usually focus on the specification of static architecture constraints rather than dynamic once.

9.3.3 Dynamic Reconfiguration Constraints

Work in this area focus on the specification of dynamic reconfiguration constraints and is most closely related to our work. However, to the best of our knowledge there exist only three approaches in this area. In the following we are going to discuss each of them in more detail.

One of the first approaches in this area is from Dormoy et al. [14] who provide a temporal logic for dynamic reconfiguration called FTPL. FTPL allows for the specification of component architecture evolution which is modeled by a transition system over architecture configurations and so-called evolution operations. While FTPL is very promising, it focuses on the temporal aspect. Thus, we complement their work by providing explicit specification techniques for data-types, interfaces and architecture configurations as well.

Castro et al. [11] provides a categorical approach to model dynamic architecture reconfigurations in terms of institutions. While the approach provides 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.

Another example is the one of Fiadeiro and Lopes [15] who provide an approach similar to ours. In their approach they use a rather abstract notion of state and configuration. While this makes the approach widely applicable, it has to be specialized for different domains. Indeed, our work can actually be seen as a specialization of their model by providing a concrete notion of state (as ports valuated by messages) and configuration (as connection of component ports).

10 Conclusion

With this article we provide a formal approach for the specification of properties for dynamic architectures by means of architecture constraints.

To this end, we first introduce an abstract model for dynamic architectures (Sect. 2). Thereby, an architecture is modeled as a set of configuration traces which are sequences over architecture configurations. An architecture configuration, on the other hand, is a set of active components and connections between their ports. A component consists of input, output, and local ports and a valuation of its ports with messages. Components are not allowed to change their interface over time, nor are they allowed to change the valuation of their local ports over time (since they act as a kind of configuration parameter).

In Sect. 3-Sect. 6 we then describe the details of our approach to specify constraints for dynamic architectures:

- First, a signature is specified defining the basic sorts, function, and predicate symbols.
- Then, datatypes are specified in terms of algebraic specification techniques over the signature (Sect. 3).
- Also interfaces are specified over the signature. An interface specification defines a set of interfaces which consist of an identifier and corresponding ports. An

interface specification allows for the specification of component types by associating interfaces with invariants formulated as over its ports (Sect. 4).
– Finally, architecture constraints are formulated over the interface specification by means of configuration assertions, i.e., linear temporal formulas over the interfaces (Sect. 6). To this end, activation and connection predicates are introduced to express activation and connection constraints, respectively.

For each specification technique a formal description of its syntax as well as its semantics in terms of our model introduced in Sect. 2 was provided.

To support in the specification process the notion of configuration diagram was introduced in Sect. 6 as a graphical notation to specify interfaces and certain common activation and connection constraints. To this end, the notion of activation and connection annotations was introduced to easily express certain common activation and connection constraints.

The approach allows to specify constraints for dynamic architectures. Therefore, it is well-suited for the specification of patterns for such architectures and enables formal analyses of such patterns as discussed in Sect. 7. This is demonstrated by a running example in which we specify the Blackboard architecture pattern and verify one of its key characteristic properties. Therefore, with our work we complement existing approaches for the specification of dynamic architectures which focus on the specification of concrete architecture instances rather than properties.

Future work arises in three major directions: (i) To support in the verification of specifications we are currently implementing of our approach for the interactive theorem prover Isabelle/HOL. (ii) On the theoretic side we are interested in a calculus of dynamic architectures to support the reasoning of dynamic architectures. (iii) Finally, we are working on an integration of our approach into current ADLs to support the specification of architecture constraints for those ADLs.

11 Acknowledgements

We would like to thank Veronika Bauer, Mario Gleirscher, Vasileios Koutsoumpas, Xiuna Zhu, and all the anonymous reviewers for their comments and helpful suggestions on earlier versions of the paper.

References

1. Gregory D Abowd, Robert Allen, and David Garlan. Formalizing style to understand descriptions of software architecture. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 4(4):319–364, 1995.
2. Robert Allen, Remi Douence, and David Garlan. Specifying and analyzing dynamic software architectures. In *Fundamental Approaches to Software Engineering*, pages 21–37. Springer, 1998.
3. Robert J Allen. A formal approach to software architecture. Technical report, DTIC Document, 1997.
4. Thais Batista, Ackbar Joolia, and Geoff Coulson. Managing dynamic reconfiguration in component-based systems. In *Proceedings of the 2Nd European Conference on Software Architecture*, EWSA'05, pages 1–17, Berlin, Heidelberg, 2005. Springer-Verlag.
5. Marco Bernardo, Paolo Ciancarini, and Lorenzo Donatiello. On the formalization of architectural types with process algebras. In *ACM SIGSOFT Software Engineering Notes*, volume 25, pages 140–148. ACM, 2000.
6. Manfred Broy. Algebraic specification of reactive systems. In *International Conference on Algebraic Methodology and Software Technology*, pages 487–503. Springer, 1996.
7. Manfred Broy. A logical basis for component-oriented software and systems engineering. *The Computer Journal*, 53(10):1758–1782, 2010.
8. Manfred Broy. A model of dynamic systems. In Saddek Bensalem, Yassine Lakhneck, and Axel Legay, editors, *From Programs to Systems. The Systems perspective in Computing*, volume 8415 of *Lecture Notes in Computer Science*, pages 39–53. Springer Berlin Heidelberg, 2014.
9. Roberto Bruni, Antonio Bucchiarone, Stefania Gnesi, Dan Hirsch, and Alberto Lluch Lafuente. Graph-based design and analysis of dynamic software architectures. In *Concurrency, Graphs and Models*, pages 37–56. Springer, 2008.
10. Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. A system of patterns: Pattern-oriented software architecture. 1996.
11. Pablo F Castro, Nazareno M Aguirre, Carlos Gustavo López Pombo, and Thomas SE Maibaum. Towards managing dynamic reconfiguration of software systems in a categorical setting. In *Theoretical Aspects of Computing–ICTAC 2010*, pages 306–321. Springer, 2010.
12. K Mani Chandy. *Parallel program design*. Springer, 1989.
13. Eric M Dashofy, André Van der Hoek, and Richard N Taylor. A highly-extensible, xml-based architecture description language. In *Software Architecture, 2001. Proceedings. Working IEEE/IFIP Conference on*, pages 103–112. IEEE, 2001.
14. Julien Dormoy, Olga Kouchnarenko, and Arnaud Lanoix. Using temporal logic for dynamic reconfigurations of components. In *Formal Aspects of Component Software*, pages 200–217. Springer, 2010.
15. José Luiz Fiadeiro and Antónia Lopes. A model for dynamic reconfiguration in service-oriented architectures. *Software & Systems Modeling*, 12(2):349–367, 2013.
16. David Garlan. Formal modeling and analysis of software architecture: Components, connectors, and events. In *Formal Methods for Software Architectures*, pages 1–24. Springer, 2003.
17. Dan Hirsch and Ugo Montanari. Two graph-based techniques for software architecture reconfiguration. *Electronic Notes in Theoretical Computer Science*, 51:177–190, 2002.
18. Charles Antony Richard Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, 1978.
19. Thomas Hungerford. *Abstract algebra: an introduction*. Cengage Learning, 2012.

20. Paola Inverardi and Alexander L Wolf. Formal specification and analysis of software architectures using the chemical abstract machine model. *Software Engineering, IEEE Transactions on*, 21(4):373–386, 1995.
21. Daniel Le Métayer. Describing software architecture styles using graph grammars. *Software Engineering, IEEE Transactions on*, 24(7):521–533, 1998.
22. David C Luckham, John J Kenney, Larry M Augustin, James Vera, Doug Bryan, and Walter Mann. Specification and analysis of system architecture using rapide. *Software Engineering, IEEE Transactions on*, 21(4):336–354, 1995.
23. Jeff Magee and Jeff Kramer. Dynamic structure in software architectures. *ACM SIGSOFT Software Engineering Notes*, 21(6):3–14, 1996.
24. Zohar Manna and Amir Pnueli. *The temporal logic of reactive and concurrent systems: Specification.* Springer Science & Business Media, 2012.
25. Diego Marmsoler and Mario Gleirscher. Specifying properties of dynamic architectures using configuration traces. In *Theoretical Aspects of Computing–ICTAC 2016*. Springer, 2016.
26. Anastasia Mavridou, Eduard Baranov, Simon Bliudze, and Joseph Sifakis. Configuration logics: Modelling architecture styles. In Christiano Braga and Peter Csaba Ölveczky, editors, *Formal Aspects of Component Software - 12th International Conference, FACS 2015, Niterói, Brazil, October 14-16, 2015, Revised Selected Papers*, volume 9539 of *Lecture Notes in Computer Science*, pages 256–274. Springer, 2015.
27. Nikunj R Mehta and Nenad Medvidovic. Composing architectural styles from architectural primitives. In *ACM SIGSOFT Software Engineering Notes*, volume 28, pages 347–350. ACM, 2003.
28. Mark Moriconi, Xiaolei Qian, and Robert A Riemenschneider. Correct architecture refinement. *Software Engineering, IEEE Transactions on*, 21(4):356–372, 1995.
29. Flavio Oquendo. π-adl: an architecture description language based on the higher-order typed π-calculus for specifying dynamic and mobile software architectures. *ACM SIGSOFT Software Engineering Notes*, 29(3):1–14, 2004.
30. John Penix, Perry Alexander, and Klaus Havelund. Declarative specification of software architectures. In *Automated Software Engineering, 1997. Proceedings., 12th IEEE International Conference*, pages 201–208. IEEE, 1997.
31. Mary Shaw and David Garlan. *Software architecture: perspectives on an emerging discipline*, volume 1. Prentice Hall Englewood Cliffs, 1996.
32. J Michael Spivey and JR Abrial. *The Z notation.* Prentice Hall Hemel Hempstead, 1992.
33. Richard N Taylor, Nenad Medvidovic, and Eric M Dashofy. *Software architecture: foundations, theory, and practice.* Wiley Publishing, 2009.
34. Michel Wermelinger and JosÃľ Luiz Fiadeiro. A graph transformation approach to software architecture reconfiguration. *Science of Computer Programming*, 44(2):133 – 155, 2002. Special Issue on Applications of Graph Transformations (GRATRA 2000).
35. Michel Wermelinger, Antónia Lopes, and José Luiz Fiadeiro. A graph based architectural (re) configuration language. In *ACM SIGSOFT Software Engineering Notes*, volume 26, pages 21–32. ACM, 2001.
36. Martin Wirsing. *Algebraic specification.* MIT Press, 1991.

12 Appendix A: Conventions

In the following we introduce some conventions used throughout the paper.

Definition 34 (Inverse function) For a function $f\colon A \to B$, we denote by $f^{-1}\colon B \to \wp(A)$, the *inverse* function of f.

Definition 35 (Bijective function) With $A \leftrightarrow B$ we denote a *bijective* function from A to B.

Definition 36 (Projection) For an n-tuple $C = (c_1, \dots, c_n)$, we denote by $[c]^i = c_i$ with $1 \leq i \leq n$ the *projection* to the i-th component of C.

Definition 37 (Partial function) We denote by $X \dashrightarrow Y$, the set of *partial* functions from a set X to a set Y.

$$X \dashrightarrow Y \overset{\text{def}}{=} \{f \subseteq X{\times}Y \mid \forall x \in X,\ y_1, y_2 \in Y\colon ((x, y_1) \in f \wedge (x, y_2) \in f) \implies y_1 = y_2\} \ .$$

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 $f\downarrow_x \overset{\text{def}}{=} f|_{\{n\in\mathbb{N} \mid n \leq x\}}$ the restriction of f to the first x numbers.

Definition 38 (Cartesian power) For a set S and number $n \in \mathbb{N}$ we denote with S^n the *cartesian power* of S to n:

$$S^n = \{(s_1, \dots, s_n) \mid s_i \in S \text{ for all } i = 1, \dots, n\} \ . \tag{29}$$

Definition 39 (Function update) For a function $f\colon D \to R$ and elements $d \in D$ and $r \in R$, we denote with $f[d \mapsto r]\colon D \to R$ a function which is equal to f but maps d to r:

$$f[d \mapsto r](x) \overset{\text{def}}{=} \begin{cases} r & \text{if } x = d \ , \\ f(x) & \text{else} \ . \end{cases} \tag{30}$$

For a family of functions $F = (F_i)_{i\in I}$ with index set I, index $j \in I$, function $F_j\colon D \to R$, elements $d \in D$ and $r \in R$, we denote by $F[j\colon d \mapsto r]$ a family where function F_j is updated to $F_j[d \mapsto r]$.

$$F[j\colon d \mapsto r]_i \overset{\text{def}}{=} \begin{cases} F_i[d \mapsto r] & \text{if } i = j \ , \\ F_i & \text{else} \ . \end{cases} \tag{31}$$

Definition 40 (Function merge) Given functions $f\colon D_f \to R_f$ and $g\colon D_g \to R_g$, with disjoint Ds and Rs, we denote with $f \cup g\colon (D_f \cup D_g) \to (R_f \cup R_g)$ the *merge* of the two functions:

$$(f \cup g)(x) \overset{\text{def}}{=} \begin{cases} f(x) & \text{if } x \in D_f \ , \\ g(x) & \text{else} \ . \end{cases} \tag{32}$$