Scientific Annals of Computer Science vol. 26 (2), 2016, pp. 187–248
doi: 10.7561/SACS.2016.2.187

On Activation, Connection, and Behavior in Dynamic Architectures

Diego Marmsoler, Mario Gleirscher[1]

Abstract

The architecture of a system describes the system's overall organization into components and connections between those components. With the emergence of mobile computing, dynamic architectures became increasingly important. In such architectures, components may appear or disappear, and connections may change over time. Despite the growing importance of dynamic architectures, the specification of properties for those architectures remains a challenge. To address this problem, we introduce the notion of *configuration traces* to model properties of dynamic architectures. Then, we characterize activation, connection, and behavior properties as special sets of configuration traces. We then show *soundness* and relative *completeness* of our characterization, i.e., we show that the intersection of an activation, connection, and behavior property contains all relevant configuration traces and that (almost) every property can be separated into these classes. Configuration traces can be used to specify general properties of dynamic architectures and the separation into different classes provides a systematic way for their specification. To evaluate our approach we apply it to the specification and verification of the Blackboard architecture pattern.

Keywords: Dynamic Architectures, Architecture Specification, Trace Semantics, Configuration traces

[1] Faculty of Computer Science, Technische Universität München, Germany, 85748 Garching bei München, E-mail: {diego.marmsoler,mario.gleirscher}@tum.de

1 Introduction

A system's architecture provides a set of components and connections between their ports. With the emergence of mobile computing, dynamic architectures became more and more important [5, 11, 24]. In such architectures, components can appear and disappear and connections can change, both over time.

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

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

A property of dynamic architectures characterizes executions of such architectures. Consider, for example, the following property for a publisher-subscriber [9] system: *Whenever a component p of type* `Publisher` *provides a message for which a* `Subscriber` *component s was subscribed, s is connected to p.* Another example describes a property of a Blackboard architecture [9]: *Whenever a component of type* `BlackBoard` *provides a message containing a problem to be solved, a component of type* `KnowledgeSource`*, able to solve this problem, is eventually activated.* Usually, such properties can be separated into different classes, such as: (i) *Behavior properties* characterizing the behavior of certain components. (ii) *Activation properties* characterizing the activation/deactivation of components. (iii) *Connection properties* characterizing the dynamic connection between components.

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

In a second step, we characterize *behavior*, *activation*, and *connection* properties as sets of configuration traces fulfilling certain closure properties. Then, we show *soundness* of our characterization, i.e., that the intersection of a behavior, activation, and a connection property contains all configuration traces satisfying corresponding activation, connection, and behavior aspects, respectively.

Moreover, we show relative completeness of our characterization. Thereby we characterize the notion of *separable architecture property* and show that each separable architecture property can be *uniquely* described through the intersection of a corresponding behavior, activation, and connection pro-

perty.

As a practical implication of our findings, we propose a method to specify dynamic architectures, separating activation, connection, and behavioral aspects. Finally, we demonstrate the approach by specifying (and verifying) the Blackboard pattern for dynamic architectures. Therefore, we first specify behavior, activation, and connection properties for Blackboard architectures. Then, we specify the pattern's guarantee as an architecture property. Finally, we verify the pattern by proving its guarantee from the original properties.

The article is organized as follows: In the remainder of this section we discuss changes to a previous version of this paper and introduce the Blackboard pattern as a running example. Section 2 introduces our model for dynamic architectures. Section 3 characterizes the different classes of properties and provides soundness and completeness results. Section 4 presents an approach to systematically specify properties and applies it to specify the Blackboard pattern. In Section 5 we provide a critical discussion of possible weaknesses of the approach. Section 6 discusses relations to existing work and Sect. 7 summarizes our results and discusses potential implications and future work. In App. A we provide full proofs of all the lemmata and theorems provided in the paper.

1.1 Previously Published Material

This paper is an extended version of Marmsoler and Gleirscher [23]. It provides improvements in the definition of some of the concepts already introduced in [23] and also provides some new concepts. In particular we introduced the crucial concept of mergeable architectures (Def. 11) and architecture merge (Def. 12) and provide a stronger version of the soundness theorem (Thm. 17). Moreover, the whole discussion of input equivalence (Sect. 2.9.4) is new. Finally, the paper provides detailed examples of all concepts and provides proofs for all lemmata and theorems.

1.2 Running Example: Specifying Blackboard Architectures

In this paper, we use the Blackboard architecture design pattern as a running example to show our approach to the specification and verification of dynamic architectures. This pattern was described, for example, by Shaw and Garlan [28], Buschmann et al. [9], and Taylor et al. [30].

Blackboards work with *problems* and *solutions* for them. Hence, we

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

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

2 A Model of Dynamic Architectures

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

2.1 Foundations

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

Convention 1 (Functions)**.** *Given two sets A and B we denote with $A \to B$ the set of functions with domain A and range B. For a function $f\colon A \to B$ we denote with* $\mathrm{dom}(f) \stackrel{def}{=} A$ *the domain of f and with* $\mathrm{ran}(f) \stackrel{def}{=} B$ *its range.*

Convention 2 (Indexed family of sets)**.** *Given a non-empty set I, we denote with $(S_i)_{i\in I}$ a family of sets indexed by I, i.e., a mapping associating a set S_i with each element $i \in I$.*

2.1.1 Messages and Ports

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

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

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

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

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

2.1.2 Valuations

In our model, components communicate by sending and receiving messages through ports. This is achieved through the notion of *port valuation*. Roughly speaking, a valuation for a set of ports is an assignment of messages to each port. Note that in our model, ports can be valuated by a set of messages meaning that a component can send/receive no message, a single message, or multiple messages at each point in time. Moreover, ports can only handle type-conform messages, i.e., messages belonging to the port's assigned type.

In the following, we denote with $\wp(S)$ the *powerset* of S, i.e., the set of all subsets of S. For ports $P \subseteq \mathsf{P}$, we denote by $\overline{P}$ the set of all type-conform *port-valuations (PVs)*, formally,

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

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

2.2 Components and Interfaces

This section introduces the basic notions of component and interface.

2.2.1 Components

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

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

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

2.2.2 Interfaces

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

Definition 3 (Component interface)**.** *A* component interface (CI) *is a pair* (P_i, P_o) *with:*

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

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

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

Interface port valuation. As for components, the same port can be used by different interfaces. Thus, to uniquely identify an interface port, we need to combine it with the corresponding interface identifier. Therefore, we can extend the notion of PV introduced in Eq. (3) for interface ports (IPRs) $P_{if} = \mathsf{I} \times \mathsf{P}$ to interface port valuations (IPVs):

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

2.3 Interface Specifications

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

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

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

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

Convention 5. *For an n-tuple* $Z = (z_1, \ldots, z_n)$*, we denote by* $\left[z\right]^i = z_i$ *with* $1 \leq i \leq n$ *the projection to the i-th component of* Z*.*

Definition 6 (Interface ports)**.** *For interface specification* $S_i = (C, I, t^c, t^i) \in \mathcal{S}_I$ *we denote by:*

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

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

The same notation can be used to denote the ports for a set of component identifiers $C' \subseteq C$ of interface specification $S_i = (C, I, t^c, t^i) \in \mathcal{S}_I$ by substituting $t^i(i)$ with $t^i(t^c(c))$ for each $c \in C'$ in the above definitions.

2.4 Specifying Interfaces

To specify interfaces, 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 the specification of architectural constraints are based on these symbols.

Then, datatypes are algebraically [8, 32] specified over the signature. A datatype specification (DTS) consists of a set of so-called datatype assertions, built over datatype terms, to assert characteristic properties of the datatype 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 by means of so-called port specifications. 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.

2.5 Running Example: Blackboard Interface Specification

A Blackboard architecture consists of a `BlackBoard` component and several `KnowledgeSource` components. Figure 1 and Fig. 2 shows corresponding datatype and port specifictaions. Figure 3 then, shows an interface specification $S_{BB} = (C, I, t^c, t^i) \in \mathcal{S}_I$ of the pattern.

DTSpec ProbSol	**imp** SET
sort PROB, SOL	
$\prec$:	PROB × PROB
solve:	PROB → SOL
well-founded($\prec$)	(4)

Figure 1: Datatype Specification.

PSpec BPort	**uses** ProbSol
i_p:	PROB × $\wp$(PROB)
i_s:	PROB × SOL
o_p:	PROB
o_s:	PROB × SOL

Figure 2: Port Specification.

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

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

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

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

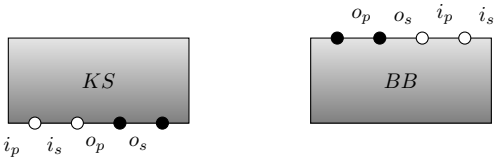


Figure 3: Interface specification for Blackboards.

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

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

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

2.6 Architecture Configurations and Configuration Traces

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

2.6.1 Architecture Configurations

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

Definition 7 (Architecture configuration)**.** *An* architecture configuration (AC) *over IS* $S_i = (C, I, t^c, t^i) \in \mathcal{S}_I$ *is a triple* (C', N, μ)*, consisting of:*

- *a set of active components* $C' \subseteq C$ *,*
- *a connection* $N \colon \mathsf{in}(S_i, C') \to \wp(\mathsf{out}(S_i, C'))$*,*
  such that $\forall (c,p) \in \mathsf{in}(S_i, C'), (c_o, p_o) \in N((c,p)) \colon T_{p_o} \subseteq T_p$ *,*
- *a valuation* $\mu \in \overline{\mathsf{port}(S_i, C')}$ *.*

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

to the corresponding connected input ports:

$$\forall \hat{p_i} \in \mathsf{in}(S_i, C') \colon N(\hat{p_i}) \neq \emptyset \implies \mu(\hat{p_i}) = \bigcup_{\hat{p_o} \in N(\hat{p_i})} \mu(\hat{p_o}) \ . \tag{5}$$

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

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

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

Convention 8. *In the following we use $c :: I$ to denote that component variable c requires any assigned component to have interface I. Moreover, port names are used to denote the corresponding port valuation.*

Example 1 (Architecture configuration)**.** *Let p_1, p_2, p_3, (p_1, s_1), $(p_2, s_2) \in \mathsf{M}$, ks_1, ks_2, $bb \in \mathsf{C}$, i_p, $i_s \in \mathsf{P_i}$, and o_p, $o_s \in \mathsf{P_o}$. Figure 4[2] shows an AC (C', N, μ) for interface specification S_{BB} (as defined in Sect. 2.5 with $C = \{ks_1, ks_2, bb\}$), with:*

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

Ports of an architecture configuration can be classified as either open or connected, depending on whether they are connected to any other ports or not. Ports which are not connected to any other port are called open configuration ports.

Definition 9 (Open configuration port)**.** *For an AC $k = (C', N, \mu) \in \mathcal{K}(S_i)$ over IS S_i we denote by:*

- $\mathsf{in}_o(S_i, k) \stackrel{def}{=} \{\hat{p} \in \mathsf{in}(S_i, C') \mid N(\hat{p}) = \emptyset\}$, *the set of* open input ports,

[2]For sake of simplicity, the configuration diagrams used in this paper only show active components of a configuration.

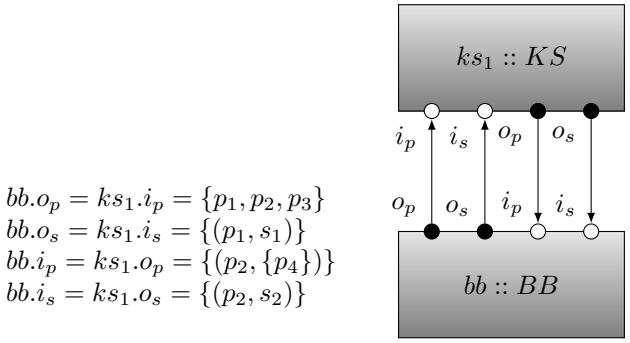


Figure 4: Architecture configuration

- $\mathsf{out}_o(S_i, k) \stackrel{def}{=} \big\{\hat{p} \in \mathsf{out}(S_i, C') \mid \nexists \hat{p}' \in \mathsf{in}(S_i, C') \colon \hat{p} \in N(\hat{p}')\big\}$, *the set of* open output ports,
- $\mathsf{port}_o(S_i, k) \stackrel{def}{=} \mathsf{in}_o(S_i, k) \cup \mathsf{out}_o(S_i, k)$, *the set of all* open ports.

On the other hand, ports which are connected to other ports are called connected configuration ports.

Definition 10 (Connected configuration port)**.** *For an AC* $k = (C', N, \mu) \in \mathcal{K}(S_i)$ *over IS* S_i *we denote by:*

- $\mathsf{in}_c(S_i, k) \stackrel{def}{=} \big\{\hat{p} \in \mathsf{in}(S_i, C') \mid N(\hat{p}) \neq \emptyset\big\}$, *the set of* connected input ports,
- $\mathsf{out}_c(S_i, k) \stackrel{def}{=} \big\{\hat{p} \in \mathsf{out}(S_i, C') \mid \exists \hat{p}' \in \mathsf{in}(S_i, C') \colon \hat{p} \in N(\hat{p}')\big\}$, *the set of* connected output ports,
- $\mathsf{port}_c(S_i, k) \stackrel{def}{=} \mathsf{in}_c(S_i, k) \cup \mathsf{out}_c(S_i, k)$, *the set of all* connected ports.

2.7 Running Example: Architecture Configurations

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

$$\forall k \in \mathcal{K}(S_{BB}), (c, p) \in \mathsf{out}(S_i, k) \colon t^c(c) = KS \implies \big[\big[k\big]^3(p)\big]^1 \in prob(c) \ . \tag{6}$$

2.8 Architecture Merge

It is possible to create a new architecture configuration from existing architecture configurations by merging activation, connection, and behavioral aspects.

In order to be mergeable, architecture configurations have to fulfill certain conditions.

Definition 11 (Mergeable architecture configurations)**.** *ACs* $k_a = (C'_a, N_a, \mu_a)$*,* $k_n = (C'_n, N_n, \mu_n)$*, and* $k_b = (C'_b, N_b, \mu_b)$ *over interface specification* $S_i \in \mathcal{S}_I$ *are called* mergeable*, denoted* $\curlyvee(k_a, k_n, k_b)$*, iff the following conditions hold:*

- *Components are consistent in their valuation of input ports:*

$$\begin{aligned}
&\forall \hat{p} \in \mathsf{in}(S_i, C'_a \cap C'_n \cap C'_b) \colon \\
&\quad \big(\hat{p} \notin \mathsf{in}(S_i, C'_a) \vee \mu_a(\hat{p}) = \emptyset\big) \ \wedge \\
&\qquad \big(\hat{p} \notin \mathsf{in}(S_i, C'_n) \vee \mu_n(\hat{p}) = \emptyset\big) \ \wedge \\
&\qquad \big(\hat{p} \notin \mathsf{in}(S_i, C'_b) \vee \mu_b(\hat{p}) = \emptyset\big) \\
&\quad \vee \mu_a(\hat{p}) = \mu_n(\hat{p}) = \mu_b(\hat{p}) \ .
\end{aligned} \tag{7}$$

- *Types and valuations of newly connected output ports are consistent with the corresponding types and valuations of connected input ports for all components:*

$$\begin{aligned}
&\forall \hat{p_i} \in \big\{\hat{p} \in \mathsf{in}(S_i, C'_a \cap C'_n) \mid N_n(\hat{p}) \cap \mathsf{out}(S_i, C'_a) \neq \emptyset\big\} \colon \\
&\quad \mu_a(\hat{p_i}) = \mu_n(\hat{p_i}) = \mu_b(\hat{p_i}) = \bigcup_{\hat{p_o} \in N_n(\hat{p_i}) \cap \mathsf{out}(S_i, C'_a) \cap \mathsf{out}(S_i, C'_b)} \mu_b(\hat{p_o}) \ .
\end{aligned} \tag{8}$$

- *AC* k_a *(responsible for activation), is not allowed to influence connection aspects:*

$$\forall (c, p) \in \mathsf{in}(S_i, C'_n), (c', p') \in N_n((c, p)) \colon c \in C'_a \wedge c' \in C'_a \ . \tag{9}$$

- *Moreover, AC* k_a *is not allowed to influence behavioral aspects:*

$$\forall (c, p) \in \mathsf{out}(S_i, C'_b) \colon \mu_b((c, p)) \neq \emptyset \implies c \in C'_a \ . \tag{10}$$

A necessary condition for architecture configurations to be mergeable is that they are consistent in their valuation of input ports as required by

Eq. (7). However, this is not a sufficient condition since an architecture merge may have newly created connected output ports, i.e., ports which are not connected in any of the original architecture configurations. Since they are to be connected in the corresponding merge, they have to be consistent in their valuation of these ports in order to satisfy Eq. (5) of Def. 7. This is what Eq. (8) requires. Finally, AC k_a, specifying the activation of the merge, is not allowed to influence neither connection nor behavior. Thus, connected components in the second AC have to be activated in the first one (9) and so has every component which has any valuations for output ports (10). In the following we provide an example of three ACs violating these requirements.

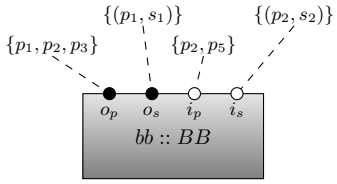
Example 2 (Non-mergeable architecture configurations)**.** *Figure 5 depicts three ACs which are not mergeable. Indeed, the three ACs violate several of the requirements provided in Def. 11.*

- *First, the ACs differ in their valuation of input ports (thus, violating Eq.* (7)*): While AC 5a has* $\{p_2, p_5\}$ *on its input port* (bb, i_p)*, the same port is valuated with* $\{p_2, \{p_4\}\}$ *in AC 5b and with* $\{p_1, p_2\}$ *in AC 5c.*
- *Moreover, since AC 5a lacks component* ks_1*, it does not allow for a merge which has connections as required from AC 5b (thus, violating Eq.* (9)*) or a behavior as required from AC 5c (thus, violating Eq.* (10)*).*
- *Finally, there is an inconsistency between AC 5b and AC 5c (thus, violating Eq.* (8)*): Since port* (ks_1, i_s) *is connected to port* (bb, o_s) *in AC 5b, the corresponding merge is required to have the same valuation of these two ports (cf. Def. 7). However, port* (bb, o_s) *is valuated by a* $\{(p_3, s_4)\}$ *in AC 5c and the corresponding input port* (ks_1, i_s) *with a* $\{(p_3, s_2)\}$*. Thus, the corresponding merge would result in an inconsistent AC.*

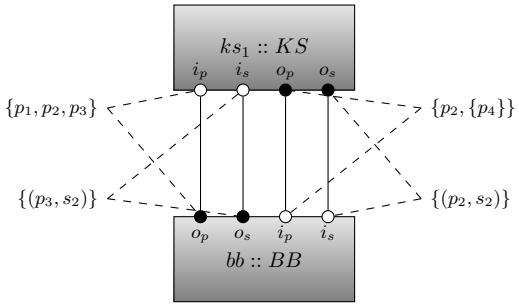
Example 3 (Mergeable architecture configurations)**.** *Figure 6 depicts three ACs which are indeed mergeable*[3]*. The ACs satisfy all conditions required by Def. 11:*

- *First, note that they all have the same valuation of input ports. Note that AC 6c has one component in addition to AC 6a. However, their input ports are all valuated by the empty set, thus, still satisfying Def. 11.*

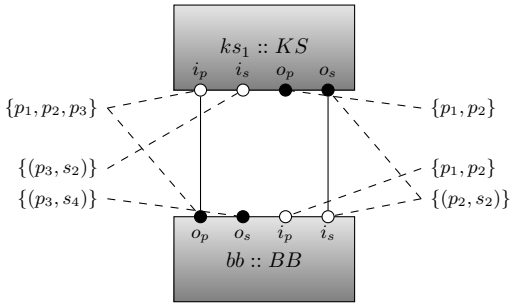
[3]Green indicates the parts of an AC which are indeed allowed to differ.



(a) First architecture configuration.



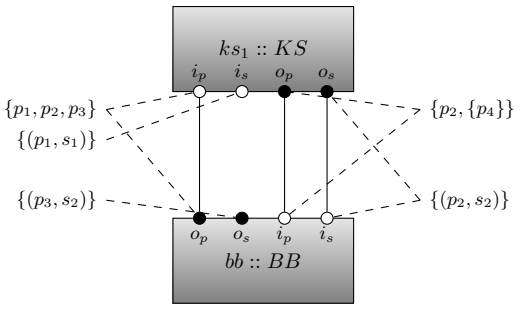
(b) Second architecture configuration.



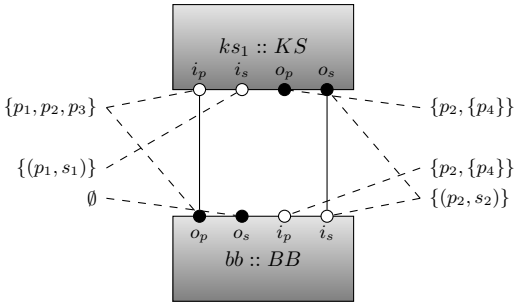
(c) Third architecture configuration.

Figure 5: Non-mergeable architecture configurations.

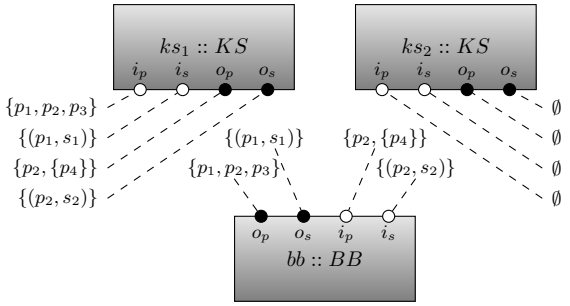
- *Note that AC 6b requires output port* (bb, o_p) *to be connected to input port* (ks_1, i_p) *and output port* (ks_1, o_s) *to be connected to input port* (bb, i_s)*. Since* (bb, o_p) *is valuated with* $\{(p_1, p_2, p_3)\}$ *and* (ks_1, o_s) *with* $\{(p_2, s_2)\}$ *in AC 6c, all three ACs are required to be valuated with* $\{(p_1, p_2, p_3)\}$ *on input port* (ks_1, i_p) *and with* $\{(p_2, s_2)\}$ *on input port* (bb, i_s)*.*
- *Finally, since AC 6c introduces a new component* ks_2 *which is not active in AC 6a, each corresponding output port must be valuated by the empty set.*



(a) First architecture configuration.



(b) Second architecture configuration.



(c) Third architecture configuration.

Figure 6: Mergeable architecture configurations.

Lemma 1 (Reflexivity of mergeable)**.** *For every AC* $k = (C', N, \mu)$*, over interface specification* $S_i \in \mathcal{S}_I$*,* k *is mergeable with itself:* $\curlyvee(k, k, k)$*.*

The proof is given in App. A.1.

Having a definition of mergeable architecture configurations allows to define the notion of architecture merge.

Definition 12 (Architecture merge)**.** *Given ACs* $k_a = (C'_a, N_a, \mu_a)$*,* $k_n = (C'_n, N_n, \mu_n)$*, and* $k_b = (C'_b, N_b, \mu_b)$ *over interface specification* $S_i \in \mathcal{S}_I$*, such that* $\curlyvee(k_a, k_n, k_b)$*, we define their* architecture merge *as the AC* $\curlyvee\big(k_a, k_n, k_b\big) = (C', N, \mu)$ *with* $C' = C'_a$*,* $N \colon \mathsf{in}(S_i, C') \to \wp(\mathsf{out}(S_i, C'))$*, such that*

$$N((c,p)) = \begin{cases} \emptyset & \text{if } c \notin C'_n \\ \{(c',p') \in N_n((c,p)) \mid c' \in C'\} & \text{if } c \in C'_n \end{cases}, \text{ and } \mu \in \overline{\mathsf{port}(S_i, C')},$$

$$\text{such that } \mu(\hat{p}) = \begin{cases} \mu_b(\hat{p}) & \text{if } \hat{p} \in \mathsf{out}(S_i, C'_b) \\ \emptyset & \text{if } \hat{p} \in \mathsf{out}(S_i, C') \setminus \mathsf{out}(S_i, C'_b) \\ \bigcup_{\hat{p_o} \in N(\hat{p})} \mu(\hat{p_o}) & \text{if } \hat{p} \in \mathsf{in}_c(S_i, \curlyvee\big(k_a, k_n, k_b\big)) \\ \mu_a(\hat{p}) & \text{if } \hat{p} \in \mathsf{in}_o(S_i, \curlyvee\big(k_a, k_n, k_b\big)) \end{cases}.$$

The architecture merge allows to create a new architecture configuration by merging the different aspects of existing architecture configurations. Thereby, a merge of three architecture configurations is a new architecture configuration with the activation of the first architecture configuration, the connection of the second architecture configuration, and the behavior of the last architecture configuration.

Example 4 (Architecture merge)**.** *Figure 7 depicts the merge of the ACs depicted in Fig. 6. The resulting AC has the same active components as*

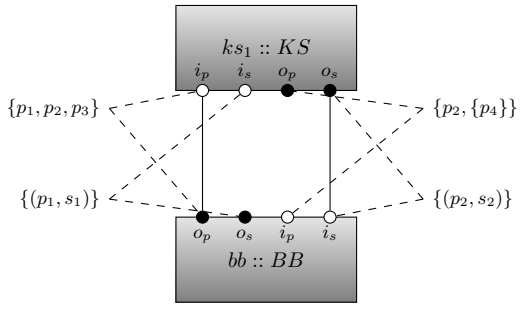


Figure 7: Merge of architecture configurations from Fig. 6.

AC 6a, the same connection as AC 6b, and the same valuation of output ports as AC 6c. Moreover, its valuation of input ports is given by the valuation of input ports of each of the original architecture configurations (which is guaranteed to be the same for each AC due to Def. 11).

Proposition 1 (Well-definedness of architecture merge)**.** *For all ACs* $k_a = (C'_a, N_a, \mu_a)$*,* $k_n = (C'_n, N_n, \mu_n)$*, and* $k_b = (C'_b, N_b, \mu_b)$ *over interface speci-*

fication $S_i \in \mathcal{S}_I$*, their architecture merge* $\curlyvee(k_a, k_n, k_b)$ *is a* unique *AC in* $\mathcal{K}(S_i)$*.*

Sketch of the proof (Full proof is given in App. A.2)**:** We show existence and uniqueness of $\curlyvee(k_a, k_n, k_b)$.

Existence: Let $\curlyvee(k_a, k_n, k_b) = (C', N, \mu)$ be defined as in Def. 12. By Def. 7 we have to show: $C' \subseteq C$, $N\colon \mathsf{in}(S_i, C') \to \wp(\mathsf{out}(S_i, C'))$, $\mu \in \overline{\mathsf{port}(S_i, C')}$, and $\forall \hat{p_i} \in \mathsf{in}(S_i, C')\colon N(\hat{p_i}) \neq \emptyset \implies \mu(\hat{p_i}) = \bigcup_{\hat{p_o} \in N(\hat{p_i})} \mu(\hat{p_o})$ to conclude $\curlyvee(k_a, k_n, k_b) \in \mathcal{K}(S_i)$.

Uniqueness: Let $k' = (C'', N', \mu')$ be defined as in Def. 12 and show $C'' = C'$, $N' = N$, and $\mu' = \mu$ to conclude $k' = \curlyvee(k_a, k_n, k_b)$.

Lemma 2 (Merge identity)**.** *For every AC* $k_a = (C'_a, N_a, \mu_a)$*, over interface specification* $S_i \in \mathcal{S}_I$*, the architecture merge of* k_a *results in* k_a *itself:* $k_a = \curlyvee(k_a, k_a, k_a)$*.*

A full proof of this lemma is given in App. A.3.

2.9 Relations Between Architecture Configurations

Architecture configurations can be related according to several aspects.

2.9.1 Activation Equivalence

One aspect which can be used to relate architecture configurations is their active components.

Definition 13 (Activation equivalence)**.** *Two ACs* $k = (C', N, \mu)$*,* $k' = (C'', N', \mu')$ *over interface specification* $S_i \in \mathcal{S}_I$*, with* $k, k' \in \mathcal{K}(S_i)$*, are* activation equivalent*, written* $k \approx^a k'$*, iff*

$$C' = C'' \ . \tag{11}$$

In the following we provide an example of two activation-equivalent architecture configurations.

Example 5 (Activation-equivalent architecture configurations)**.** *Fig. 8 depicts two activation-equivalent ACs. They both have active components* ks_1 *and* bb*. Note that they differ in their valuation of ports* (ks_1, o_p) *and* (bb, i_p)*. Moreover, they differ in their connection since in one of them* (ks_1, o_p) *is connected to* (bb, i_p) *while in the other these ports are unconnected.*

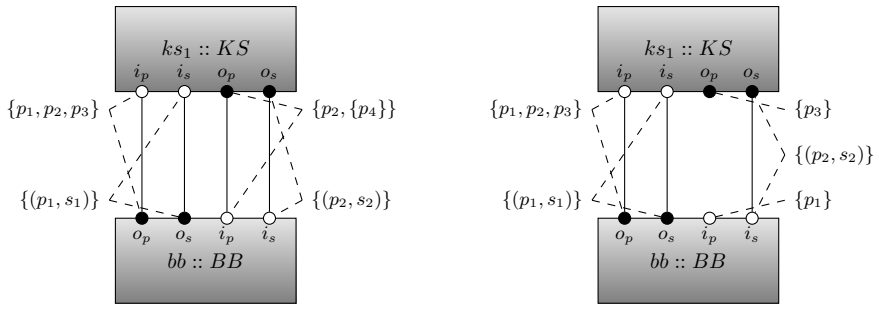


Figure 8: Two activation-equivalent architecture configurations.

We also provide an example of two architecture configurations which are not activation equivalent.

Example 6 (Non-activation-equivalent architecture configurations)**.** *Fig. 9 depicts two ACs which are not considered to be activation equivalent. While*

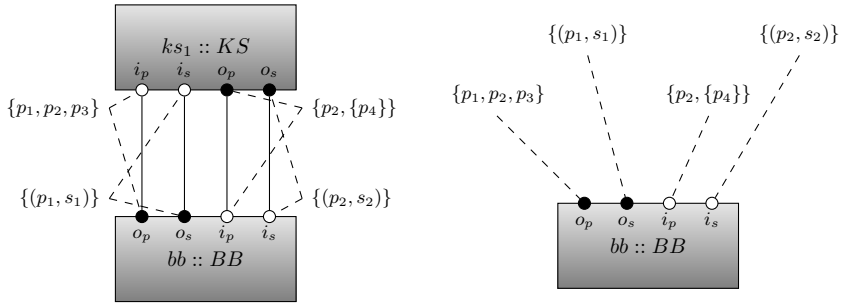


Figure 9: Two non-activation-equivalent architecture configurations.

component ks_1 is active in one of the ACs, it is deactivated in the other. Note that they are equal, however, in all their port-valuations.

Note that activation equivalence is indeed an equivalence relation.

Lemma 3 (Activation equivalence)**.** *Activation equivalence is an equivalence relation.*

Sketch of the proof (Full proof is given in App. A.4)**:** It is reflexive, symmetric, and transitive.

Finally, an architecture merge is always activation equivalent to its first architecture configuration.

Lemma 4 (Activation equivalence of merge)**.** *For all ACs* $k_a = (C'_a, N_a, \mu_a)$, $k_n = (C'_n, N_n, \mu_n)$, *and* $k_b = (C'_b, N_b, \mu_b)$ *over interface specification* $S_i \in \mathcal{S}_I$ *we have that their merge* $\curlyvee(k_a, k_n, k_b)$ *is activation equivalent to* k_a*:* $\curlyvee(k_a, k_n, k_b) \approx^a k_a$.

A full proof of this lemma is given in App. A.5.

2.9.2 Connection Equivalence

Another aspect according to which architecture configurations can be related is their connections.

Definition 14 (Connection equivalence)**.** *Two ACs* $k = (C', N, \mu)$ *and* $k' = (C'', N', \mu')$ *over interface specification* $S_i \in \mathcal{S}_I$ *are* connection equivalent*, written* $k \approx^n k'$*, iff*

$$\begin{aligned}
&\forall \hat{p_i} \in \mathsf{in}(S_i, C' \cap C'') \colon N(\hat{p_i}) \;= N'(\hat{p_i}) \;\wedge \\
&\forall \hat{p_i} \in \mathsf{in}(S_i, C' \setminus C'') \colon N(\hat{p_i}) \;= \emptyset \;\wedge \\
&\forall \hat{p_i} \in \mathsf{in}(S_i, C'' \setminus C') \colon N'(\hat{p_i}) = \emptyset \;\;.
\end{aligned}$$

Note that the deactivation of components is interpreted as if all their ports are not connected to any other port. Thus, if an architecture configuration k has a component activated which is not activated in another architecture configuration k', all the ports of this component have to be unconnected in k, in order for k to be (possibly) connection equivalent to k'.

In the following we provide an example of two connection-equivalent architecture configurations.

Example 7 (Connection-equivalent architecture configurations)**.** *Fig. 10 depicts two connection-equivalent ACs. Note that they do have two input ports in common:* i_p *and* i_s *of component bb and that their connection agrees on these input ports, i.e., they are unconnected in both ACs. Moreover,* i_p *and* i_s *of component* ks_1 *are only available in the first AC. Thus, Def. 14 requires them to be unconnected.*

Note also that both ACs differ in their active components (ks_1 *is only active in one of the two) and in their valuation of ports, e.g.,* (bb, o_s) *and* (bb, i_s).

We also provide an example of two architecture configurations which are not connection equivalent.

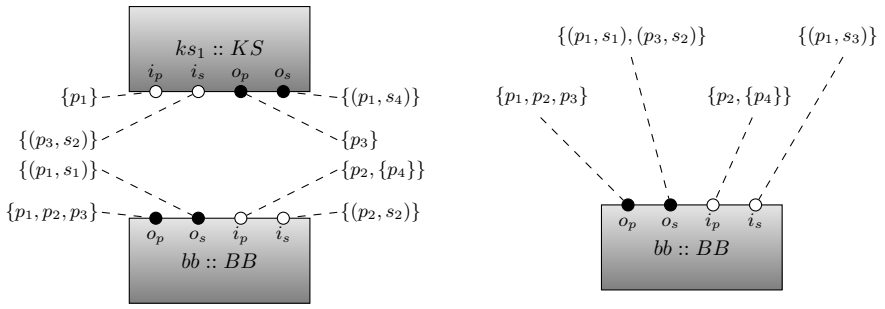


Figure 10: Two connection-equivalent architecture configurations.

Example 8 (Non-connection-equivalent architecture configurations)**.** *Fig. 11 depicts two ACs which are not considered to be connection equivalent. Alt-*

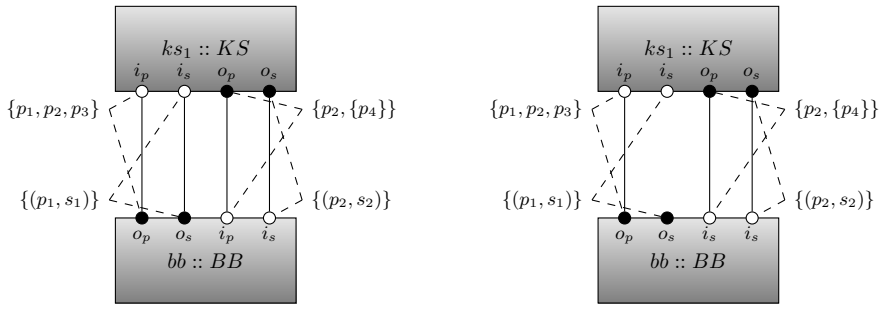


Figure 11: Two non-connection-equivalent architecture configurations.

hough both ACs have ks_1 and bb activated, in one of the ACs, port (ks_1, i_s) is connected to port (bb, o_s) while in the other one, these two ports are unconnected. Note, however, that they are equal in their components as well as their valuation of ports.

In the following we provide an important property of connection equivalence.

Lemma 5 (Connection equivalence)**.** *Connection equivalence is an equivalence relation.*

Sketch of the proof (Full proof is given in App. A.6)**:** It is reflexive, symmetric, and transitive.

An important property of an architecture merge is that it is always connection equivalent to its second architecture configuration.

Lemma 6 (Connection equivalence of merge)**.** *For all ACs* $k_a = (C'_a, N_a, \mu_a)$, $k_n = (C'_n, N_n, \mu_n)$, *and* $k_b = (C'_b, N_b, \mu_b)$ *over interface specification* $S_i \in \mathcal{S}_I$ *we have that their merge* $\curlyvee\big(k_a, k_n, k_b\big)$ *is connection equivalent to* k_n*:* $\curlyvee\big(k_a, k_n, k_b\big) \approx^n k_n$.

A full proof of this lemma is provided in App. A.7.

2.9.3 Behavior Equivalence

Architecture configurations can also be related according to their behavior, i.e., the valuation of their output ports.

Definition 15 (Behavior equivalence)**.** *Two ACs* $k = (C', N, \mu)$ *and* $k' = (C'', N', \mu')$ *over interface specification* $S_i \in \mathcal{S}_I$ *are* behavior equivalent*, written* $k \approx^b k'$*, iff*

$$\begin{aligned}
&\forall \hat{p} \in \mathsf{out}(S_i, C' \cap C'') \colon \mu(\hat{p}) \;= \mu'(\hat{p}) \;\wedge \\
&\forall \hat{p} \in \mathsf{out}(S_i, C' \setminus C'') \colon \mu(\hat{p}) \;= \emptyset \;\wedge \\
&\forall \hat{p} \in \mathsf{out}(S_i, C'' \setminus C') \colon \mu'(\hat{p}) = \emptyset \;.
\end{aligned}$$

Similar to connection equivalence, the deactivation of components is interpreted as if all their output ports are valuated by the empty set. Thus, if an architecture configuration k has a component activated which is not activated in another architecture configuration k', all the output ports of this component have to be valuated by the empty set in k, in order for k to be (possibly) behavior equivalent to k'.

In the following we provide an example of two behavior equivalent architecture configurations.

Example 9 (Equivalent architecture configurations)**.** *Fig. 12 depicts two behavior equivalent ACs. Note that they do indeed differ in their active components as well as their valuation of input ports. However, as required by Def. 15, they do have the same valuation of their common output ports* (bb, o_p) *and* (bb, o_s)*. Moreover, all output ports which exist only in the first AC (*o_p *and* o_s *of component* ks_1*) are valuated by the empty set.*

We also provide an example of two architecture configurations which are not behavior equivalent.

Example 10 (Non-equivalent architecture configurations)**.** *Fig. 13 depicts two ACs which are not considered to be behavior equivalent. Although both*

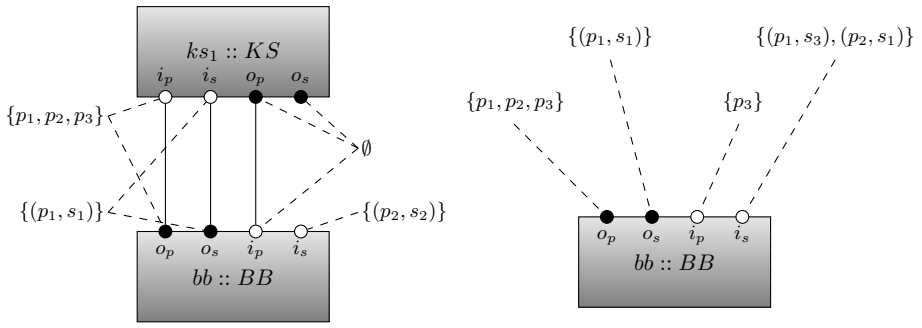


Figure 12: Two behavior equivalent architecture configurations.

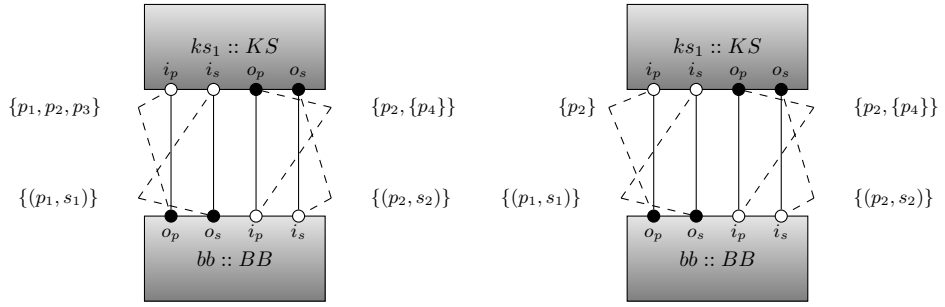


Figure 13: Two non behavior equivalent architecture configurations.

ACs have the same components activated and the same connections between their ports, in one of the ACs, port (bb, o_p) *is valuated with* $\{p_1, p_2, p_3\}$*, while in the other AC the same port is valuated with* $\{p_2\}$*.*

Also behavior equivalence is indeed an equivalence relation.

Lemma 7 (Behavior equivalence)**.** *Behavior equivalence is an equivalence relation.*

The proof for this lemma is similar to the proof of Lem. 5.

An important property of an architecture merge is that it is always behavior equivalent to its third architecture configuration.

Lemma 8 (Behavior equivalence of merge)**.** *For all ACs* $k_a = (C'_a, N_a, \mu_a)$*,* $k_n = (C'_n, N_n, \mu_n)$*, and* $k_b = (C'_b, N_b, \mu_b)$ *over interface specification* $S_i \in \mathcal{S}_I$ *we have that their merge* $\curlyvee(k_a, k_n, k_b)$ *is behavior equivalent to* k_b*:* $\curlyvee(k_a, k_n, k_b) \approx^b k_b$*.*

A full proof of this lemma is provided in App. A.8.

2.9.4 Input Equivalence

Finally, architecture configurations can be related according to their valuation of input ports. Note that we separate input equivalence from behavior (output) equivalence here. The reason is, that all the aspects of dynamic architectures (including behavior) are usually specified based on the valuation of input ports.

Definition 16 (Input equivalence)**.** *Two ACs* $k = (C', N, \mu)$, $k' = (C'', N', \mu')$ *over interface specification* $S_i \in \mathcal{S}_I$*, with* $k, k' \in \mathcal{K}(S_i)$*, are* input equivalent*, written* $k \approx^i k'$*, iff*

$$\begin{aligned}
\forall \hat{p} \in \mathsf{in}(S_i, C' \cap C'') \colon \mu(\hat{p}) &= \mu'(\hat{p}) \ \wedge \\
\forall \hat{p} \in \mathsf{in}(S_i, C' \setminus C'') \colon \mu(\hat{p}) &= \emptyset \ \wedge \\
\forall \hat{p} \in \mathsf{in}(S_i, C'' \setminus C') \colon \mu'(\hat{p}) &= \emptyset \ .
\end{aligned}$$

Again, the deactivation of components is interpreted as if all their input ports are valuated by the empty set. Thus, if an architecture configuration k has a component activated which is not activated in another architecture configuration k', all the input ports of this component have to be unconnected in k, in order for k to be (possibly) input equivalent to k'.

In the following we provide an example of two input equivalent architecture configurations.

Example 11 (Equivalent architecture configurations)**.** *Fig. 14 depicts two input equivalent ACs. Note that they do indeed differ in their active components as well as their valuation of output ports. However, as required by Def. 16, they do have the same valuation of their common input ports*

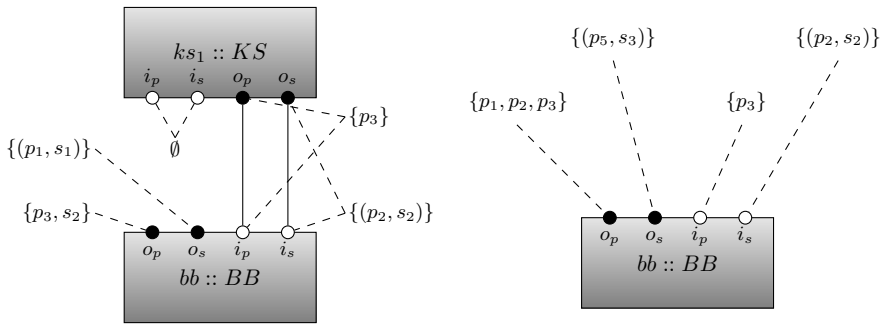


Figure 14: Two input-equivalent architecture configurations.

(bb, i_p) and (bb, i_s). Moreover, all input ports which exists only in the first AC (i_p and i_s of component ks_1) are valuated by the empty set.

We also provide an example of two architecture configurations which are not input equivalent.

Example 12 (Non-equivalent architecture configurations)**.** *Fig. 15 depicts two ACs which are not considered to be input equivalent. Although both ACs*

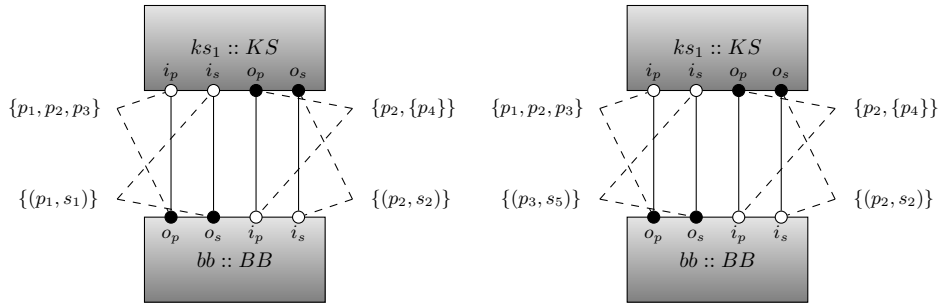


Figure 15: Two non-input-equivalent architecture configurations.

have the same components activated and the same connections between their ports, in one of the ACs, port (ks_1, i_s) is valuated with $\{(p_1, s_1)\}$, while in the other AC the same port is valuated with $\{(p_3, s_5)\}$.

Also input equivalence is indeed an equivalence relation.

Lemma 9 (Input equivalence)**.** *Input equivalence is an equivalence relation.*

Sketch of the proof (Full proof is given in App. A.9)**:** Proof is similar to the one for Lem. 7.

A property of mergeable architecture configurations is that they are indeed input equivalent to each other.

Lemma 10 (Input equivalence of mergeable)**.** *For all ACs $k_a, k_n, k_b \in \mathcal{K}(S_i)$, such that $\curlyvee(k_a, k_n, k_b)$, we have $k_a \approx^i k_n$, $k_a \approx^i k_b$, and $k_n \approx^i k_b$.*

A full proof of this lemma is provided in App. A.10.

An important property of an architecture merge is that it is always input equivalent to all its architecture configurations.

Lemma 11 (Input equivalence of merge)**.** *For all ACs $k_a, k_n, k_b \in \mathcal{K}(S_i)$, such that $\curlyvee(k_a, k_n, k_b)$, each AC is input equivalent to their merge $\curlyvee\big(k_a, k_n, k_b\big)$:*

$$\begin{aligned} k_a &\approx^i \curlyvee\big(k_a, k_n, k_b\big) \ , \\ k_n &\approx^i \curlyvee\big(k_a, k_n, k_b\big) \ , \ \textit{and} \\ k_b &\approx^i \curlyvee\big(k_a, k_n, k_b\big) \ . \end{aligned}$$

Sketch of the proof (Full proof is given in App. A.11)**:** $k_a \approx^i \curlyvee\big(k_a, k_n, k_b\big)$: For open input ports equality follows from Def. 12. Equality of connected input ports follows from Def. 11. $k_n \approx^i \curlyvee\big(k_a, k_n, k_b\big)$ and $k_b \approx^i \curlyvee\big(k_a, k_n, k_b\big)$ follows from Lem. 10 and transitivity of $\approx^i$ (Lem. 9).

2.9.5 Relating Activation, Connection, Behavior, and Input

The relations introduced so far suffice to determine architecture configuration equivalence.

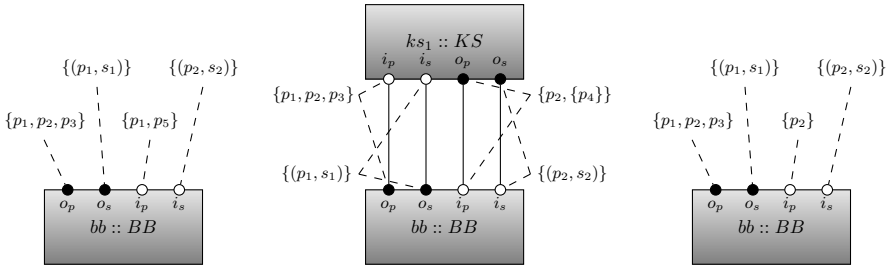
Lemma 12 (Equality of architecture configurations)**.** *Two ACs $k, k' \in \mathcal{K}(S_i)$ are the same iff they are activation equivalent, connection equivalent, behavior equivalent, and input equivalent:*

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

The proof of this lemma is given in App. A.12.

Note, that the right hand side of the equation in Lem. 12 is *not* the weakest one to determine equality of architecture configurations. Indeed, it would be sufficient to require that k and k' have the same valuation of their *open* input ports. The equality of connected input ports follows then from $k \approx^b k'$ and $k \approx^n k'$ since Def. 7 requires connected input ports to be valuated by the union of the valuation of the corresponding output ports. Nevertheless, the above equality is sufficient for the following explanations.

Open input equivalence. The above observation would actually provide evidence to introduce a *weaker notion of input equivalence* in which we only required *open* input ports to be equivalent in order for two architecture configurations to be considered input equivalent. As it turns out, however, this is *not sufficient to achieve transitivity* which is an important property required for the remaining discussion.



(a) First AC. (b) Second AC. (c) Third AC.

Figure 16: Three architecture configurations.

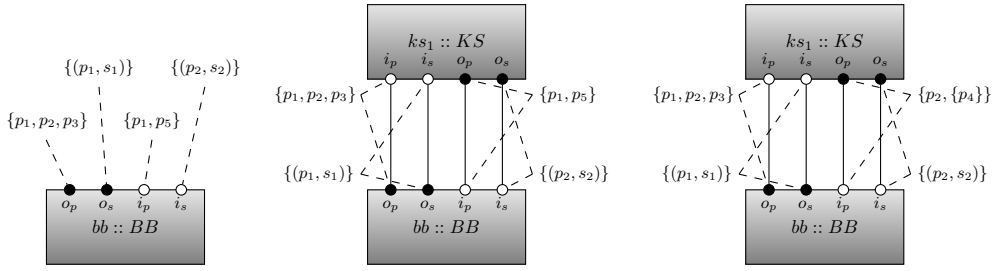
Example 13 (Why equivalence of open input ports is not transitive)**.** *Consider the three ACs depicted in Fig. 16. Since AC 16a and AC 16b have no common* open *input ports, they have vacuously the same valuation of their common open input ports. Similar for AC 16b and AC 16c. However, AC 16a and AC 16c share two* open *input ports i_p and i_s and they differ in the valuation of one of them.*

Therefore, if input equivalence would only consider valuations of open input ports, AC 16a and AC 16b would be considered input equivalent and so would be AC 16b and AC 16c. However, AC 16a and AC 16c would not be considered input equivalent which is why this notion of input equivalence would not be transitive and, thus, not an equivalence relation at all.

Strong equivalence of open input ports. We could even try to further strengthen the notion of equivalence of open input ports to consider all input ports which are *open* in at least one of the two architecture configurations. Although this is now a stronger version as the one discussed in Ex. 13 (indeed architecture configuration 16a and architecture configuration 16b or architecture configuration 16b and architecture configuration 16c would not be considered input equivalent anymore), it is still weaker than our working definition of input equivalence provided by Def. 16.

However, as shown in the following example, also this notion of input equivalence is not transitive.

Example 14 (Why strong equivalence of open input ports is not transitive)**.** *Consider the three ACs depicted in Fig. 17. Since ports i_p and i_s of component bb are* open *input ports in AC 16a, they need to be considered when relating another AC to AC 16a. However, since the valuation of these two*



(a) First AC. (b) Second AC. (c) Third AC.

Figure 17: Three architecture configurations.

ports is the same for AC 16a and AC 16b, they would be considered input equivalent. AC 16b and AC 16c have no open input ports, at all. Therefore, also these two would be considered input equivalent. However, AC 16c varies in its valuation of input port (bb, i_p) *from AC 16a which is why AC 16a and AC 16c would not be considered input equivalent.*

Thus, if input equivalence would also consider valuations of input ports which are open *in one of the ACs, AC 16a and AC 16b would be considered input equivalent and so would be AC 16b and AC 16c. However AC 16a and AC 16c would not be considered input equivalent which is why also this notion of input equivalence would not be transitive and thus not an equivalence relation, at all.*

Thus, although Def. 16 is not the weakest definition of input equivalence which leads to Lem. 12, it is the only one which is transitive.

2.10 Configuration Traces

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

Definition 17 (Configuration trace)**.** *A* configuration trace (CT) *over interface specification* $S_i \in \mathcal{S}_I$ *is a mapping* $\mathbb{N} \to \mathcal{K}(S_i)$*. The set of all CTs for* S_i *is denoted by* $\mathcal{K}^t(S_i)$*.*

Example 15 (Configuration trace)**.** *Figure 18 shows a CT* $t \in \mathcal{K}^t(S_i)$ *with corresponding ACs* $t(0) = k_0$*,* $t(1) = k_1$*, and* $t(2) = k_2$*. AC* k_0*, for example, is shown in Example 1.*

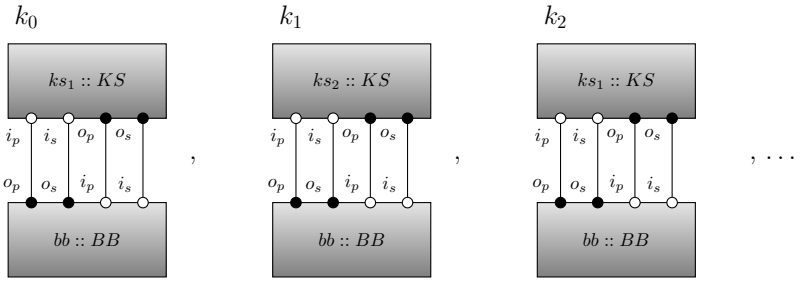


Figure 18: Configuration trace (port valuations not shown, see Fig. 4 for an example)

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

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

- *components* may appear and disappear over time and
- *connections* may change over time.

We can lift the corresponding definitions for architecture configurations to configuration traces.

Definition 18 (Equivalences and mergeable for configuration traces)**.** *Given two CTs $t, t' \in \mathcal{K}^t(S_i)$ over interface specification S_i we have*

$$t \approx^a t' \overset{def}{\iff} \forall n \in \mathbb{N}\colon t(n) \approx^a t'(n) \ ,$$
$$t \approx^n t' \overset{def}{\iff} \forall n \in \mathbb{N}\colon t(n) \approx^n t'(n) \ ,$$
$$t \approx^b t' \overset{def}{\iff} \forall n \in \mathbb{N}\colon t(n) \approx^b t'(n) \ , \ \textit{and}$$
$$t \approx^i t' \overset{def}{\iff} \forall n \in \mathbb{N}\colon t(n) \approx^i t'(n) \ .$$

Given other architecture traces $t'', t''' \in \mathcal{K}^t(S_i)$ we have

$$\curlyvee(t', t'', t''') \overset{def}{\iff} \forall n \in \mathbb{N}\colon \curlyvee\left(t'(n), t''(n), t'''(n)\right)$$

and

$$\curlyvee\big(t', t'', t'''\big) \overset{def}{=} \lambda n \in \mathbb{N}\colon \curlyvee\big(t'(n), t''(n), t'''(n)\big) \ .$$

2.11 Specifying Configuration Traces

Configuration traces can be specified by means of configuration trace assertions formulated over interface specifications. Configuration diagrams are a graphical extension which allow to graphically express certain configuration trace assertions by annotating a given interface specification.

In the following, we provide a brief, informal description of these techniques. A more detailed discussion including a presentation of the formal semantics of these techniques is provided in [21].

2.11.1 Configuration Trace Assertions

Configuration trace assertions (CTAs) are a temporal specification technique based on linear temporal logic [20] to specify sets of CTs. They are based on the notion of configuration assertions (CNFAs) which allow for the specification of a components state. Roughly speaking, CNFAs are formulæspecified over a components interface. Thereby, port names denote the valuation of the corresponding component port within an architecture configuration and can be used as variables in algebraic terms as well. For example, $c.p$ denotes the current valuation of port p of component c. Moreover, CNFAs allow for the specification of activation and connection predicates:

- *Activation predicates* can be used to specify activation and deactivation of components. An activation of component c, for example, is denoted with $\|c\|$.
- *Connection predicates* can be used to specify connection between component ports. A connection between port p of component c and port p' of component c', for example, is denoted with $c.p \rightarrow c'.p'$.

CTAs are then build over CNFAs. Thus, a CNFA is itself a CTA. Moreover, if ϕ and ψ are CTAs, then $\Diamond\phi$, $\Box\phi$, $\bigcirc\phi$, and $\phi\ \mathcal{W}\ \psi$ are CTAs with the following meaning:

- $\Diamond\phi$ holds iff ϕ holds of at least one configuration in t,
- $\Box\phi$ holds iff ϕ holds of all configurations in t,
- $\bigcirc\phi$ holds iff ϕ holds of the following configuration in t,
- $\phi\ \mathcal{W}\ \psi$ holds iff ϕ holds of all configurations before ψ holds or iff $\Box\phi$.

2.12 Configuration Diagrams

A configuration diagram (CD) is a graph whose nodes resemble interfaces (group of ports) and whose edges denote connections between component ports. CDs can be annotated by certain activation and connection constraints:

- *Activation annotations* can be used to introduce common activation constraints, such as min./max. number of components of a certain type.
- *Connection annotations* can be used to denote connection constraints, such as required connections between components of a certain type.

3 Specifying Properties of Dynamic Architectures

Properties of dynamic architectures can be specified as sets of configuration traces over an interface specification. In the following, we investigate the nature of such properties and introduce the notion of *behavior*, *activation*, and *connection* properties. We then show that the intersection of such properties is guaranteed to contain all necessary configuration traces. Moreover, we introduce the notion of *separable architecture property* and show that such a property can always be uniquely represented as the intersection of corresponding behavior, activation, and connection properties.

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

3.1 Activation Properties

An set of CTs is an activation property if it does neither restrict behavior nor connection.

Definition 19 (Activation property)**.** *An* activation property (AP) *for interface specification $S_i \in \mathcal{S}_I$ is a set of CTs $A \subseteq \mathcal{K}^t(S_i)$, such that connections and behavior are not restricted:*

$$\forall t_a \in A, t_n, t_b \in \mathcal{K}^t(S_i) \colon \curlyvee(t_a, t_n, t_b) \\ \implies \exists t'_a \in A \colon t'_a \approx^a t_a \wedge t'_a \approx^n t_n \wedge t'_a \approx^b t_b \wedge t'_a \approx^i \curlyvee\big(t_a, t_n, t_b\big) \ .$$

Thus, activation properties are defined by means of a special closure property for a set of configuration traces A. It requires that for each confi-

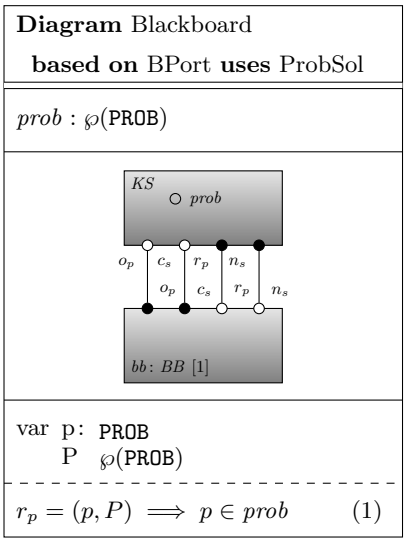


Figure 19: Configuration diagram for Blackboard.

guration trace in A, there exists an activation-equivalent configuration trace in A for every possible connection and behavior.

3.1.1 Running Example: Activation Property Specification.

Activation properties of the Blackboard pattern are described in the *configuration diagram* [21] in Fig. 19: The "[1]" annotation for blackboard interfaces (e.g. BB) denotes the condition that components have to be active from the beginning on whereas knowledge source interfaces (e.g. KS) allow corresponding components to be de-/activated over time.

Moreover, we require two additional properties specified in terms of CTAs [21], a temporal-logic notation (based on [20]) to specify sets of configuration traces.

Figure 20 shows the corresponding specification. We require that architectures are fair w.r.t. knowledge source activation, i.e., each knowledge source is activated infinitely many times (Eq. 12). Furthermore, we require that whenever a knowledge source offers to solve some problem, it is always activated when solutions to the required subproblems are provided (Eq. 13).

Note that the *activation constraints* induced by the diagram in Fig. 19 as well as the additional constraints specified in Fig. 20 constrain only the activation of components. They do neither restrict connections nor behavior

Spec Blackboard_Activation **uses** *Blackboard*

var bb : BB
ks : KS
$p,\ q$: PROB
P : PROB SET

$$\Box\Big(\|ks\| \implies \bigcirc\big(\Diamond\|ks\|\big)\Big) \tag{12}$$
$$\forall(p,P) \in ks.o_p \colon \forall q \in P \colon \Box\Big((q, solve(q)) \in bb.o_s \Rightarrow \|ks\|\Big) \tag{13}$$

Figure 20: Specification of activation constraints for Blackboard architectures.

which is why the resulting architecture property is indeed an example of an activation property as defined in Def. 19.

3.2 Connection Properties

A connection property is not allowed to restrict either behavior or activation.

Definition 20 (Connection property)**.** *A* connection property (CP) *for interface specification $S_i \in \mathcal{S}_I$ is a set of CTs $N \subseteq \mathcal{K}^t(S_i)$, such that activations and behavior are not restricted:*

$$\forall t_n \in N, t_a, t_b \in \mathcal{K}^t(S_i) \colon \curlyvee(t_a, t_n, t_b)$$
$$\implies \exists t'_n \in N \colon t'_n \approx^n t_n \wedge t'_n \approx^a t_a \wedge t'_n \approx^b t_b \wedge t'_n \approx^i \curlyvee\big(t_a, t_n, t_b\big) \ .$$

Thus, connection properties are defined similar to activation properties, by means of a special closure property for a set of configuration traces N. It requires that for each configuration trace in N, there exists a connection-equivalent configuration trace in A for every possible activation and behavior.

3.2.1 Running Example: Connection Property Specification

Connection properties are also specified graphically in the configuration diagram in Fig. 19. The solid arcs denote a constraint requiring that the ports of a `KnowledgeSource` component are connected with the corresponding ports of a `BlackBoard` component as depicted, whenever both components are active.

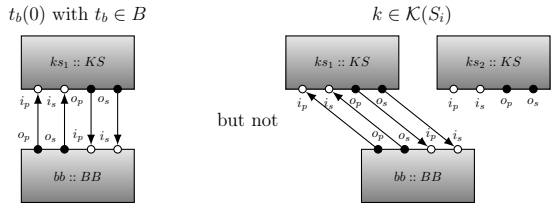


Figure 21: Example of an ill-formed behavioral property.

Note that the connection constraints induced by the diagram in Fig. 19 constrain only the connection of components. They do neither restrict activation nor behavior. Thus, the resulting architecture property is indeed an example of a connection property as defined in Def. 20.

3.3 Behavior Properties

A behavioral property is an architecture property which does not constrain connections and activations.

Definition 21 (Behavior property)**.** *A* behavioral property (BP) *for an interface specification $S_i = (C, I, t^c, t^i) \in \mathcal{S}_I$ is a set of CTs $B \subseteq \mathcal{K}^t(S_i)$, such that connections and activations are not restricted:*

$$\forall t_b \in B, t_a, t_n \in \mathcal{K}^t(S_i)\colon \curlyvee(t_a, t_n, t_b)$$
$$\implies \exists t'_b \in B\colon t'_b \approx^b t_b \wedge t'_b \approx^a t_a \wedge t'_b \approx^n t_n \wedge t'_b \approx^i \curlyvee\big(t_a, t_n, t_b\big)\ .$$

Again, behavioral properties are defined by means of a special closure property for a set of configuration traces B. It requires that for each configuration trace in B, there exists a behavior-equivalent configuration trace in B for every possible activation and connection.

Example 16 (Not a behavior property)**.** *Figure 21 shows how an architecture property B can violate Def. 21: Assume that B allows a CT t_b with $t_b(0)$ and denies some traces with k with $k \approx^b t_b(0)$ at $n = 0$, i.e. $\nexists t'_b \in B : t'_b(0) \approx^a k \vee t'_b(0) \approx^n k$. Hence, B constrains activation and, thus, wrongly contains parts of an AP.*

3.3.1 Running Example: Behavior Property Specification

We provide behavioral properties for both, `BlackBoard` and `KnowledgeSource` components. Again, we specify the properties in terms of CTAs. Thereby,

variables denote component identifiers, problems and solutions. Port names are used to denote port valuations and $c :: I$ is used to denote that component identifier c has interface I.

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

The specification for `BlackBoard` components is given in Fig. 22. We

Spec Blackboard_Behavior **uses** *Blackboard*

var bb : BB
$p,\ p'$: PROB
P : PROB SET
s : SOL

$$\Box\Big((p,s) \in bb.i_s \implies \Diamond\big((p,s) \in bb.o_s\big)\Big) \tag{14}$$
$$\Box\Big((p,P) \in bb.i_p \implies \big(\forall p' \in P\colon\ (\Diamond p' \in bb.o_p)\big)\Big) \tag{15}$$
$$\Box\Big(p \in bb.o_p \implies \big(p \in bb.o_p\ \mathcal{W}\ (p, solve(p)) \in bb.i_s\big)\Big) \tag{16}$$

Figure 22: Specification of behavior for blackboard components.

require three properties for them:

Eq. (14)**:** If a solution to a subproblem is received on its input, then it is eventually provided at its output.

Eq. (15)**:** If solving a problem requires a set of subproblems to be solved first, those problems are eventually provided at its output.

Eq. (16)**:** A problem is provided as long as it is not solved.

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

The specification for `KnowledgeSource` components is given in Fig. 23. Also for them we require three properties:

Spec Knowledgesource_Behavior **uses** *Blackboard*

var ks : KS
$p,\ q$: PROB
P : PROB SET

$$\Box\Big(\forall(p,P)\in ks.o_p\colon\ \big((\forall q\in P\colon \Diamond(q,solve(q))\in ks.i_s)\Longrightarrow\Diamond(p,solve(p))\in ks.o_s\big)\Big) \quad (17)$$
$$\Box\Big(\forall(p,P)\in ks.o_p\colon \forall q\in P\colon q\prec p\Big) \quad (18)$$
$$\Box\Big(p\in ks.prob\wedge p\in ks.i_p\implies\Diamond(\exists P\colon (p,P)\in ks.o_p)\Big) \quad (19)$$

Figure 23: Specification of behavior for knowledgesource components.

Eq. (17)**:** If a `KnowledgeSource` gets correct solutions for all the required subproblems, then it solves the problem eventually.

Eq. (18)**:** In order to solve a problem, a `KnowledgeSource` requires solutions only for smaller problems:

Eq. (19)**:** If a `KnowledgeSource` is able to solve a problem it will eventually communicate this:

Note that Eq. (14)-(19) constrain only the behavior of components. They do neither restrict activation nor connections. Thus, the resulting architecture property is indeed an example of a behavioral property as defined in Def. 21.

3.4 Soundness of Property Characterization

In the following, we show *soundness* of our characterization. Therefore, we show that the intersection of an activation, connection, and behavioral property contains all necessary configuration traces (and not more).

First, however, we provide an important property necessary to prove the soundness theorem. It states that for every three configuration traces, there exists a unique configuration trace which is activation, connection, and behavior equivalent to the original trace.

Lemma 13 (Uniqueness)**.** *For all $t_a, t_n, t_b \in \mathcal{K}^t(S_i)$, such that $\curlyvee(t_a, t_n, t_b)$, there exists a* unique *$t \in \mathcal{K}^t(S_i)$, such that $t \approx^a t_a$, $t \approx^n t_n$, $t \approx^b t_b$, and $t \approx^i \curlyvee\big(t_a, t_n, t_b\big)$.*

Sketch of the proof (Full proof is given in App. A.15)**:** *Existence*: Construct $t \in \mathcal{K}^t(S_i)$, such that for all $n \in \mathbb{N}$: $t(n) \stackrel{\text{def}}{=} \curlyvee\big(t_a(n), t_n(n), t_b(n)\big)$. *Uniqueness*: By Lem. 12.

Now we have everything to formulate and prove soundness of our characterization.

Theorem 17 (Soundness)**.** *Let $B \subseteq \mathcal{K}^t(S_i)$ be a BP, $N \subseteq \mathcal{K}^t(S_i)$ a CP, and $A \subseteq \mathcal{K}^t(S_i)$ an AP. Then, for every $t \in \mathcal{K}^t(S_i)$,*

$$t \in A \cap N \cap B \iff \exists t_a \in A, t_n \in N, t_b \in B\colon \curlyvee(t_a, t_n, t_b) \wedge t \approx^a t_a \wedge t \approx^n t_n \wedge t \approx^b t_b \wedge t \approx^i \curlyvee\big(t_a, t_n, t_b\big) \ .$$

Sketch of the proof (Full proof is given in App. A.16)**:**

- $\Longrightarrow$: Let $t_a = t_n = t_b = t$ and show $t_a \in A \wedge t_n \in N \wedge t_b \in B$, $t_a \approx^i t_n \wedge t_a \approx^i t_b \wedge t_n \approx^i t_b$, $t \approx^a t_a \wedge t \approx^n t_n \wedge t \approx^b t_b$, and $t \approx^i \curlyvee\big(t_a, t_n, t_b\big)$.
- $\Longleftarrow$: Since $t_a \in A$, $t_n, t_b \in \mathcal{K}^t(S_i)$, and $t_a \approx^i t_n \wedge t_a \approx^i t_b \wedge t_n \approx^i t_b$, have $\exists t'_a \in A$ such that $t'_a \approx^a t_a$, $t'_a \approx^n t_n$, $t'_a \approx^b t_b$, and $t'_a \approx^i \curlyvee\big(t_a, t_n, t_b\big)$ by Def. 19 and conclude $t'_a = t \in A$ by Lem. 13. Similar reasoning can be applied to show $t \in N$ and $t \in B$.

3.5 Completeness of Property Characterization

In addition to soundness, we provide also a completeness result for our characterization. Therefore, we show that (almost) every set of configuration traces can be separated into an activation, connection, and behavioral property.

Indeed, completeness does not hold for every set of configuration traces. In the following, we characterize those sets for which it does.

3.5.1 Separable Architecture Properties

A separable architecture property is a set of configuration traces where the aspect (activation, connection, and behavior) are independent from each other.

Definition 22 (Separable architecture property)**.** *A* separable architecture property (SAP) *for interface specification $S_i = (C, I, t^c, t^i) \in \mathcal{S}_I$, is a set of CTs $S \subseteq \mathcal{K}^t(S_i)$, such that activation, connection, and behavior do not influence each other:*

$$\forall t \in \mathcal{K}^t(S_i), n \in \mathbb{N}\colon \Big(\big(\exists t_a \in S\colon t_a \approx^a t \wedge t_a \approx^i t\big) \ \wedge \big(\exists t_n \in S\colon t_n \approx^n t \wedge t_n \approx^i t\big) \wedge \big(\exists t_b \in S\colon t_b \approx^b t \wedge t_b \approx^i t\big)\Big) \implies t \in S \ .$$

Also separable architecture properties are defined by means of a special closure property: for each separable architecture properties S we require that for each configuration trace t, if there exist traces t_a, t_n, and t_b, such that t_a is activation equivalent to t, t_n is connection equivalent to t, and t_b is behavior equivalent to t, then t is also in S. In the following, we provide an example of a separable architecture properties.

Example 18 (Separable architecture property)**.** *Figure 24 depicts the idea behind SAPs. If there exists an AC which is activation, connection, and*

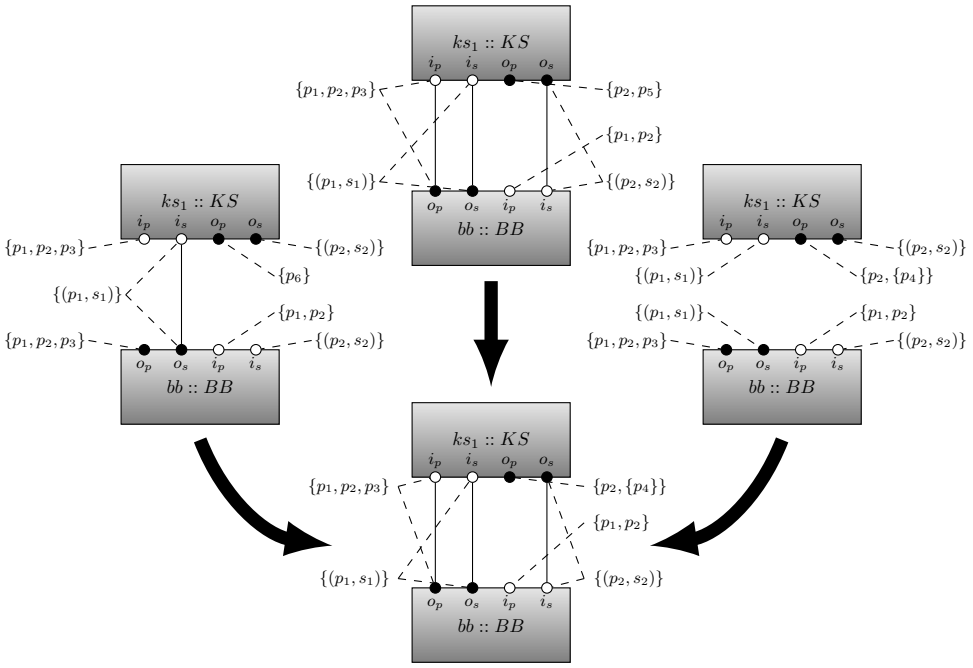


Figure 24: Separable architecture property.

behavior equivalent to three other ACs which are part of the SAP, then the original AC must also be part of the SAP. In Fig. 24 the AC at the bottom is activation equivalent to the left AC, connection equivalent to the top AC, and behavior equivalent to the right AC. If the top three ACs are part of a SAP, then the bottom one has to be part of the SAP, too.

3.5.2 Running Example: Blackboard Guarantee

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

Theorem 19 (Blackboard)**.** *If for each open (sub-)problem, there exists a Knowledge-source (KS) which is able to solve the corresponding problem:*

$$\square\Big(\forall p \in bb.o_p \colon \Diamond\big(\exists ks \colon p \in ks.prob \wedge \|ks\|\big)\Big) \ , \tag{20}$$

then, it is guaranteed, that the architecture will eventually solve an overall problem, even if no single KS is able to solve the problem on its own:

$$\square\Big(p \in bb.i_p \implies \Diamond(p, solve(p)) \in bb.o_s\Big) \ . \tag{21}$$

Proof: (Sketch) The proof is by well-founded induction over the problem relation $\prec$: We are sure that for each problem eventually a `KnowledgeSource` exists which is capable to solve the problem, Eq. (20). The required subproblems are provided to the `BlackBoard` by the connection constraint of Fig. 19. The `BlackBoard` will provide these subproblems eventually on its output o_p, Eq. (15). Since the subproblems provided to the `BlackBoard` are strictly less (according to $\prec$), Eq. (18), they will be solved and provided by the `BlackBoard` by induction. A `KnowledgeSource` will eventually be activated for each solution, Eq. (20), and connected to the `BlackBoard` (Fig. 19). This `KnowledgeSource` eventually has all solutions to its subproblems, by Eqs. (12) and (13), and will then solve the original problem by Eq. (17). The solution is received eventually by the `BlackBoard` due to Fig. (19). Finally, the overall solution is provided by the `BlackBoard` due to Eq. (14). □

3.5.3 Relative Completeness

In order to show relative completeness of our characterization we have to introduce the notion of activation, connection, and behavior closures for a set of configuration traces.

An activation closure takes a set of configuration traces and adds all activation equivalent configuration traces.

Definition 23 (Activation closure)**.** *For a set of CTs $A \subseteq \mathcal{K}^t(S_i)$ over interface specification $S_i \in \mathcal{S}_I$ an activation closure $\mathsf{close}^{\mathsf{a}}(A) \subseteq \mathcal{K}^t(S_i)$ is defined as follows:*

$$\mathsf{close}^{\mathsf{a}}(A) = \left\{t' \in \mathcal{K}^t(S_i) \mid \exists t \in A \colon t \approx^a t' \wedge t \approx^i t'\right\} \ .$$

An important property for an activation closure is that it is indeed an activation property.

Lemma 14 (Activation closure is an activation property)**.** *For a set of CTs* $A \subseteq \mathcal{K}^t(S_i)$ *over interface specification* $S_i \in \mathcal{S}_I$*,* $\mathsf{close}^{\mathsf{a}}(A)$ *is an AP.*

Sketch of the proof (Full proof is given in App. A.13)**:** Let $t_a \in \mathsf{close}^{\mathsf{a}}(A)$ and $t_n, t_b \in \mathcal{K}^t(S_i)$, such that $\curlyvee(t_a, t_n, t_b)$ and construct $t'_a \in \mathcal{K}^t(S_i)$, such that for all $n \in \mathbb{N}$, $t'_a(n) \stackrel{\text{def}}{=} \curlyvee\big(t_a(n), t_n(n), t_b(n)\big)$. Then, show that $t'_a \approx^a t_a$, $t'_a \approx^n t_n$, $t'_a \approx^b t_b$, and $t'_a \approx^i \curlyvee\big(t_a, t_n, t_b\big)$.

Similar to the activation closure, we can introduce the notion of connection closure $\mathsf{close}^{\mathsf{n}}(T) \subseteq \mathcal{K}^t(S_i)$ and behavior closure $\mathsf{close}^{\mathsf{b}}(T) \subseteq \mathcal{K}^t(S_i)$ for a set of CTs $T \subseteq \mathcal{K}^t(S_i)$ over interface specification $S_i \in \mathcal{S}_I$. A similar lemma as Lem. 14 can then be proved for these notions.

Now we have everything we need to proof a completeness theorem for our characterization of activation, connection, and behavior properties.

Theorem 20 (Completeness)**.** *Each SAP* $S \subseteq \mathcal{K}^t(S_i)$ *for interface specification* $S_i \in \mathcal{S}_I$ *can be* uniquely *described through the intersection of an AP* $A \subseteq \mathcal{K}^t(S_i)$*, CP* $N \subseteq \mathcal{K}^t(S_i)$*, and BP* $B \subseteq \mathcal{K}^t(S_i)$*:*

$$S = A \cap N \cap B \ .$$

Sketch of the proof (Full proof is given in App. A.14)**:** $A \cap N \cap B \subseteq S$: Apply $\mathsf{close}^{\mathsf{a}}(S)$, $\mathsf{close}^{\mathsf{n}}(S)$, $\mathsf{close}^{\mathsf{b}}(S)$ to construct A, N, and B, respectively. Then show that their intersection is a subset of S by applying Def. 22. $A \cap N \cap B \supseteq S$: Use reflexivity of $\approx^a$, $\approx^n$, and $\approx^b$ to conclude $A \cap N \cap B \supseteq S$ by Def. 23.

4 Verifying Properties of Dynamic Architectures

In this section, we propose an approach to the verification of properties for dynamic architectures based on the theory discussed so far.

Figure 25 shows the proposed approach to property verification. In a first step, components interfaces are specified. Based on the interface specification corresponding behavior, connection, and activation properties are specified. Finally, an overall architecture property is specified and verified against the behavior, connection, and activation properties.

In the following we discuss the details of specification and verification steps.

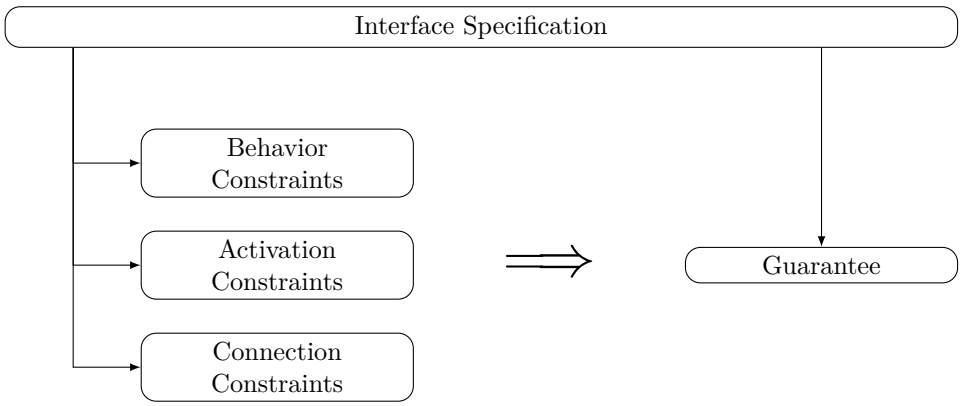


Figure 25: Verification approach.

4.1 Specifying Interfaces

To specify interfaces, a set of ports and corresponding types of messages have to be specified first. This can be achieved by traditional specification techniques such as algebraic specifications [32]. Interfaces can then be simply specified by grouping a set of ports.

4.2 Specifying Activation, Connection, and Behavior Properties

Activation, connection, and behavior properties can then be specified over the interfaces. Fig. 26 depicts an overview of the proposed approach to specify architecture properties. After specifying activation, connection, and behavior properties, the specifications are merged by taking the intersection of the corresponding sets of configuration traces.

The different kind of properties can be specified by CTAs [21]. However, since they allow to specify general sets of CTs, they have to be formulated in a manner such that the corresponding definitions are satisfied. A general heuristics could be, for example, to use implications of the following form:

$$\text{General CTA} \implies \text{Property Specific CTA} \ .$$

Thereby, the left hand side of the implication is a general CTA while the right hand side is one, specific for the kind of property one wants to specify. These property-specific CTAs are only allowed to specify activation, connection, or behavior, respectively.

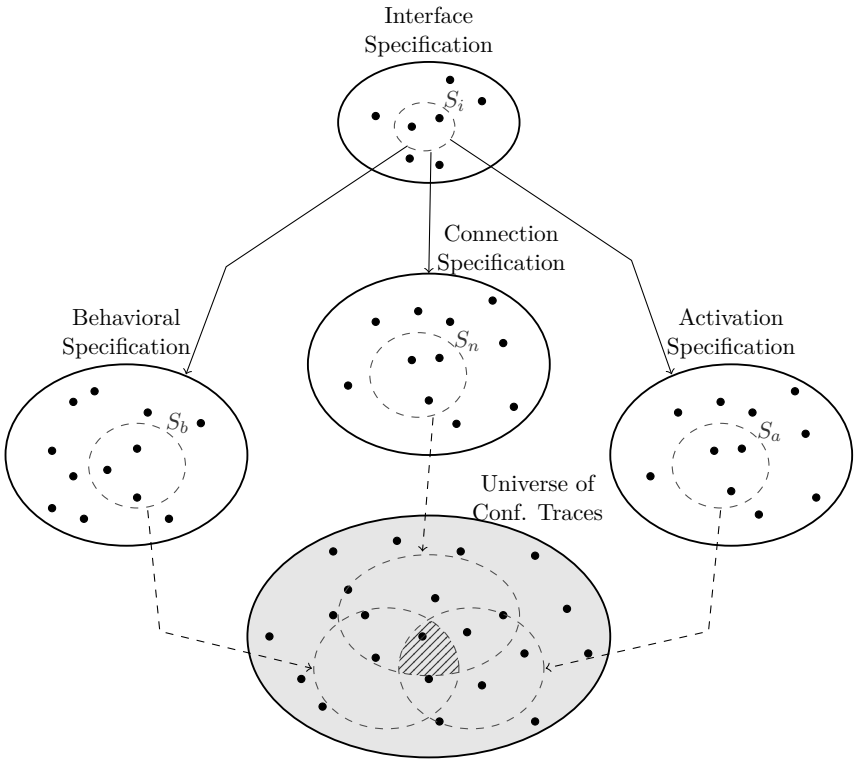


Figure 26: Specifying Dynamic Architectures.

4.3 Verifying the Guarantee

The guarantee is specified as general architecture property over the corresponding interface specification. Again, CTAs can be used to specify the guarantee. The guarantee is then verified by showing that the corresponding set of configuration traces is a subset of the set specified as the intersection of the activation, connection, and behavior properties.

5 Discussion

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

Theoretical limitations. One possible weakness concerns the nature of our underlying model for dynamic architectures. Def. 17 does not allow components to change their interface over time. This could be seen as a restriction of the model, however, it was a deliberate decision since for now, we did not yet find the need for components to change their interfaces. Indeed, it remains an open question whether dynamic interfaces are useful, at all. However, if the need for them arises, it should be noted, that the underlying model can be adapted to allow for dynamic interfaces as well.

Methodological limitations. With this text, we provide a formal characterization of three different classes of properties for dynamic architectures. Moreover, with Thm. 17 and Thm. 20 we provide evidence that these classes are indeed useful. To verify that a property is indeed of a certain class, one has to show that it satisfies the corresponding definition. We admit that this could be seen as a potential weakness in that it is not always obvious whether a given property indeed satisfies one of the characteristic definitions. However, the purpose of this text was to provide a clear understanding of these classes which serves as a basis for future work concentrating on how to support in the verification of whether a property indeed belongs to one of the different classes.

Practical limitations. A last point which needs to be discussed in more detail regards an important aspect of software architectures in general. Our approach does actually not provide means to directly specify quality attributes such as availability or reliability. However, as our example shows, it allows us to specify the technical realization of such aspects. Theorem 3.5.2, for example, ensures that a problem can be solved also in the absence of certain components. This can be seen as one possible implementation (or technical definition) of what is sometimes called availability.

6 Related Work

Related work can be found in three different areas: 1. Architecture description languages, 2. Modeling of architectural styles, and 3. Specification of constraints for dynamic architectures. In the following we briefly discuss each of them.

6.1 Architecture Description Languages

Over the last three decades, a number of so-called Architecture Description Languages (ADLs) emerged to support the formal specification of architectures. Some of them also support the specification of dynamic aspects such as Rapide [18], Darwin [19], Dynamic Wright [2, 3], Π-ADL [26], xADL [12], and ACME [14].

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

6.2 Modeling Architectural Styles

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

One of the first approaches to formalize architectural styles is discussed by Abowd et al. [1]. There, the authors apply a denotational semantics approach to software architectures by using the specification language Z [29]. Other examples used to specify architectural styles include the Chemical Abstract Machine [16] or Wright [2] which allow for the specification of architectural constraints for static architectures. Two further ideas come from Moriconi et al. [25] and Penix et al. [27]. Both apply algebraic specification to software architectures. Finally, Bernardo et al. [4] use process algebras to specify architectural types which can be seen as a form of architectural styles.

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

6.3 Specification of Constraints for Dynamic Architectures

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

The approach of Le Métayer [17] applies graph theory to specify architectural evolution. The author proposes the use of graph grammars to specify architectural evolution. A similar approach comes from Hirsch and

Montanari [15] who employ hypergraphs as a formal model to represent styles and their reconfigurations. While we also apply a graph-based approach to model architectural properties, the major difference lies in the specification of behavior. While the discussed approaches focus on structural aspects, we aim at a combination of structural and behavioral aspects.

Another, closely related approach is the one of Wermlinger et al. [31]. The authors combine behavior and structure to model dynamic reconfigurations. One major difference to our work concerns the underlying model of interaction. While the authors use an action synchronization communication model, our model is based on time-synchronous communication. Both communication models have their advantages and drawbacks. Thus, by providing a time-synchronous alternative, we actually complement their work.

Recently, categorical approaches to dynamic architecture reconfiguration appeared such as the work of Castro et al. [10] or Fiadeiro and Lopes [13]. While these approaches provide fundamental insights into the specification of dynamic architecture properties, their model remains implicit in the categorical constructions. Thus, we complement their work by providing an explicit model of dynamic architecture properties.

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

7 Conclusion

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

- We provide a *novel model for dynamic architectures* and a formal notion of *properties* for this kind of architectures (Sect. 2). Thereby an architecture property is modeled as a set of configuration traces (Def. 17) which are sequences of architecture configurations (Def. 7).
- We provide a formal characterization of *activation properties*, *connection properties*, and *behavior properties* for dynamic architectures (Sect. 3). Each property-type is defined as a set of configuration traces fulfilling a special closure property: An activation property is not

allowed to restrict connections or behavior (Def. 19). A connection property, on the other hand, is neither allowed to restrict activation nor behavior (Def. 20). Finally, a behavioral property is not allowed to restrict activation or connection (Def. 21).
- We show soundness of our characterization. Thereby, we show that the intersection of activation, connection, and behavior properties include exactly all expected configuration traces (Thm. 17).
- Finally, we provide a completeness result for our characterization. Therefore we provide a formal characterization of *separable architecture properties* and show that each separable architecture property can be represented as the intersection of a corresponding activation, connection, and behavior properties (Thm. 20).

Our results can be used to specify and verify properties of dynamic architectures. Therefore, we derive a systematic way to specify properties for dynamic architectures and apply it to the specification of blackboard architectures (Sect. 4):

- First, component interfaces are specified as a set of ports and corresponding types.
- Then, activation, connection, and behavior properties can be specified over the interfaces.
- Then, a guarantee can be specified as a general architecture property.
- Finally, the guarantee is verified by showing that it is a subset of the intersection of the corresponding activation, connection, and behavior properties.

The approach is demonstrated in terms of a running example in which the blackboard pattern for dynamic architectures is specified and verified.

With this work we provide an important step towards our overall goal of providing a formal theory of architectural patterns [22]. Future work should now concentrate around two major areas:

- One area of future work concerns the practical usability of the approach. As discussed in Sect. 5, future work should concentrate on the development of advanced techniques for the specification and analysis of the different classes of properties.
- A second area of future work concerns the application of the approach. As demonstrated by the running example, the approach is well-suited for the verification of patterns for dynamic architectures. Thus, future work in this area should concentrate on the application of the approach for the purpose of verifying existing patterns of dynamic architectures.

Acknowledgments

We would like to thank Jonas Eckhardt, Vasileios Koutsoumpas, and the anonymous reviewers of ICTAC 2016 and Scientific Annals of Computer Science for their comments and helpful suggestions. This work was partially funded by the German Federal Ministry of Education and Research (BMBF) under grant 01Is16043A.

References

[1] G. D. Abowd, R. Allen, and D. Garlan. "Formalizing Style to Understand Descriptions of Software Architecture". In: *ACM Transactions on Software Engineering and Methodology (TOSEM)* 4.4 (1995), pp. 319–364. DOI: 10.1145/226241.226244.

[2] R. J. Allen. *A Formal Approach to Software Architecture.* Tech. rep. DTIC Document, 1997.

[3] R. Allen, R. Douence, and D. Garlan. "Specifying and Analyzing Dynamic Software Architectures". In: *Fundamental Approaches to Software Engineering*. Ed. by E. Astesiano. Vol. 1382. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 1998, pp. 21–37. DOI: 10.1007/bfb0053581.

[4] M. Bernardo, P. Ciancarini, and L. Donatiello. "On the Formalization of Architectural Types with Process Algebras". In: *ACM SIGSOFT Software Engineering Notes*. Vol. 25. 6. 2000, pp. 140–148. DOI: 10.1145/357474.355064.

[5] J. S. Bradbury et al. "A survey of self-management in dynamic software architecture specifications". In: *Proceedings of the 1st ACM SIGSOFT workshop on Self-managed systems - WOSS '04*. ACM Press, 2004, pp. 28–33. DOI: 10.1145/1075405.1075411.

[6] M. Broy. "A Logical Basis for Component-Oriented Software and Systems Engineering". In: *The Computer Journal* 53.10 (Feb. 2010), pp. 1758–1782. DOI: 10.1093/comjnl/bxq005.

[7] M. Broy. "A Model of Dynamic Systems". In: *From Programs to Systems. The Systems perspective in Computing*. Ed. by S. Bensalem, Y. Lakhneck, and A. Legay. Vol. 8415. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2014, pp. 39–53. ISBN: 978-3-642-54847-5. DOI: 10.1007/978-3-642-54848-2_3.

[8] M. Broy. "Algebraic Specification of Reactive Systems". In: *Algebraic Methodology and Software Technology*. Springer Berlin Heidelberg, 1996, pp. 487–503. DOI: 10.1007/bfb0014335.

[9] F. Buschmann et al. *Pattern-Oriented Software Architecture: A System of Patterns*. Wiley West Sussex, England, 1996.

[10] P. F. Castro et al. "Towards Managing Dynamic Reconfiguration of Software Systems in a Categorical Setting". In: *Lecture Notes in Computer Science*. Springer, 2010, pp. 306–321. DOI: 10.1007/978-3-642-14808-8_21.

[11] P. C. Clements. "A survey of architecture description languages". In: *Proceedings of the 8th International Workshop on Software Specification and Design*. IEEE Comput. Soc. Press, 1996, p. 16. DOI: 10.1109/iwssd.1996.501143.

[12] E. M. Dashofy, A. Van der Hoek, and R. N. Taylor. "A Highly-Extensible, XML-Based Architecture Description Language". In: *Software Architecture, 2001. Proceedings. Working IEEE/IFIP Conference on*. 2001, pp. 103–112. DOI: 10.1109/wicsa.2001.948416.

[13] J. L. Fiadeiro and A. Lopes. "A Model for Dynamic Reconfiguration in Service-oriented Architectures". In: *Software & Systems Modeling* 12.2 (2013), pp. 349–367. DOI: 10.1007/s10270-012-0236-1.

[14] D. Garlan. "Formal Modeling and Analysis of Software Architecture: Components, Connectors, and Events". In: *Formal Methods for Software Architectures*. Springer, 2003, pp. 1–24. DOI: 10.1007/978-3-540-39800-4_1.

[15] D. Hirsch and U. Montanari. "Two Graph-Based Techniques for Software Architecture Reconfiguration". In: *Electronic Notes in Theoretical Computer Science* 51 (May 2002), pp. 177–190. DOI: 10.1016/s1571-0661(04)80201-9.

[16] P. Inverardi and A. L. Wolf. "Formal Specification and Analysis of Software Architectures Using the Chemical Abstract Machine Model". In: *Software Engineering, IEEE Transactions on* 21.4 (1995), pp. 373–386. DOI: 10.1109/32.385973.

[17] D. Le Métayer. "Describing Software Architecture Styles Using Graph Grammars". In: *Software Engineering, IEEE Transactions on* 24.7 (1998), pp. 521–533. DOI: 10.1109/32.708567.

[18] D. C. Luckham et al. “Specification and Analysis of System Architecture Using Rapide”. In: *Software Engineering, IEEE Transactions on* 21.4 (1995), pp. 336–354. DOI: 10.1109/32.385971.

[19] J. Magee and J. Kramer. “Dynamic Structure in Software Architectures”. In: *ACM SIGSOFT Software Engineering Notes* 21.6 (1996), pp. 3–14. DOI: 10.1145/239098.239104.

[20] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems*. Springer New York, 1992. DOI: 10.1007/978-1-4612-0931-7.

[21] D. Marmsoler. “On the Specification of Constraints for Dynamic Architectures”. In: *ArXiv e-prints* (Mar. 2017). arXiv: 1703.06823 [cs.SE].

[22] D. Marmsoler. “Towards a Theory of Architectural Styles”. In: *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering - FSE 2014*. ACM Press, 2014, pp. 823–825. DOI: 10.1145/2635868.2661683.

[23] D. Marmsoler and M. Gleirscher. “Specifying Properties of Dynamic Architectures using Configuration Traces”. In: *Theoretical Aspects of Computing*. Springer, 2016. DOI: 10.1007/978-3-319-46750-4_14.

[24] N. Medvidovic. “ADLs and dynamic architecture changes”. In: *Joint proceedings of the second international software architecture workshop (ISAW-2) and international workshop on multiple perspectives in software development (Viewpoints ’96) on SIGSOFT ’96 workshops -*. ACM Press, 1996, pp. 24–27. DOI: 10.1145/243327.243340.

[25] M. Moriconi, X. Qian, and R. A. Riemenschneider. “Correct Architecture Refinement”. In: *Software Engineering, IEEE Transactions on* 21.4 (1995), pp. 356–372. DOI: 10.1109/32.385972.

[26] F. Oquendo. “π-ADL: An Architecture Description Language based on the Higher-Order Typed π-Calculus for Specifying Dynamic and Mobile Software Architectures”. In: *ACM SIGSOFT Software Engineering Notes* 29.3 (May 2004), pp. 1–14. DOI: 10.1145/986710.986728.

[27] J. Penix, P. Alexander, and K. Havelund. “Declarative Specification of Software Architectures”. In: *Automated Software Engineering*. 1997, pp. 201–208. DOI: 10.1109/ase.1997.632840.

[28] M. Shaw and D. Garlan. *Software Architecture: Perspectives on an Emerging Discipline*. Vol. 1. Prentice Hall Englewood Cliffs, 1996. ISBN: 9780131829572.

[29] J. Spivey. *The Z Notation: A Reference Manual*. Prentice-Hall international series in computer science. Prentice Hall, 1992. ISBN: 9780139785290.

[30] R. N. Taylor, N. Medvidovic, and E. M. Dashofy. *Software Architecture: Foundations, Theory, and Practice*. Wiley Publishing, 2009. ISBN: 9780470167748.

[31] M. Wermelinger, A. Lopes, and J. L. Fiadeiro. "A Graph Based Architectural (Re)configuration Language". In: *Software Engineering Notes*. Vol. 26. 5. 2001, pp. 21–32. DOI: 10.1145/503271.503213.

[32] M. Wirsing. "Algebraic Specification". In: *Handbook of Theoretical Computer Science (Vol. B)*. Ed. by J. van Leeuwen. Cambridge, MA, USA: MIT Press, 1990, pp. 675–788. ISBN: 0-444-88074-7. DOI: 10.1016/b978-0-444-88074-1.50018-4.

A Proofs

A.1 Proof of Lem. 1

According to Def. 11 we have to show that (i) $\forall \hat{p} \in \mathsf{in}(S_i, C') \colon (\hat{p} \notin \mathsf{in}(S_i, C') \vee \mu(\hat{p}) = \emptyset) \vee \mu(\hat{p}) = \mu(\hat{p})$, (ii) $\forall \hat{p} \in \mathsf{in}(S_i, C') \colon N(\hat{p}) \cap \mathsf{out}(S_i, C') \neq \emptyset \implies \mu(\hat{p}) = \bigcup_{\hat{p_o} \in N(\hat{p}) \cap \mathsf{out}(S_i, C')} \mu(\hat{p_o})$, (iii) $\forall (c, p) \in \mathsf{in}(S_i, C'), (c', p') \in N((c, p)) \colon c \in C' \wedge c' \in C'$, and (iv) $\forall (c, p) \in \mathsf{out}(S_i, C') \colon \mu((c, p)) \neq \emptyset \implies c \in C'$:

(i) $\forall \hat{p} \in \mathsf{in}(S_i, C') \colon (\hat{p} \notin \mathsf{in}(S_i, C') \vee \mu(\hat{p}) = \emptyset) \vee \mu(\hat{p}) = \mu(\hat{p})$: Since $\forall \hat{p} \in \mathsf{in}(S_i, C') \colon \mu(\hat{p}) = \mu(\hat{p})$.

(ii) $\forall \hat{p} \in \mathsf{in}(S_i, C') \colon N(\hat{p}) \cap \mathsf{out}(S_i, C') \neq \emptyset \implies \mu(\hat{p}) = \bigcup_{\hat{p_o} \in N(\hat{p}) \cap \mathsf{out}(S_i, C')} \mu(\hat{p_o})$: Let $\hat{p} \in \mathsf{in}(S_i, C')$ and assume $N(\hat{p}) \cap \mathsf{out}(S_i, C') \neq \emptyset$. Thus, conclude $N(\hat{p}) \neq \emptyset$ and have $\mu(\hat{p}) = \bigcup_{\hat{p_o} \in N(\hat{p})} \mu(\hat{p_o}) = \bigcup_{\hat{p_o} \in N(\hat{p}) \cap \mathsf{out}(S_i, C')} \mu(\hat{p_o})$ by Def. 7.

(iii) $\forall (c, p) \in \mathsf{in}(S_i, C'), (c', p') \in N((c, p)) \colon c \in C' \wedge c' \in C'$: By Def. 7.

(iv) $\forall (c, p) \in \mathsf{out}(S_i, C') \colon \mu((c, p)) \neq \emptyset \implies c \in C'$: Since $\forall (c, p) \in \mathsf{out}(S_i, C') \colon c \in C'$ by Def. 6. □

A.2 Proof of Prop. 1

Existence: Let $\curlyvee(k_a, k_n, k_b) = (C', N, \mu)$ with $C' = C'_a$, $N \colon \mathsf{in}(S_i, C') \to \wp(\mathsf{out}(S_i, C'))$, such that $N((c, p)) = \begin{cases} \emptyset & \text{if } c \notin C'_n \\ \{(c', p') \in N_n((c, p)) \mid c' \in C'\} & \text{if } c \in C'_n \end{cases}$, and $\mu \in \overline{\mathsf{port}(S_i, C')}$, such that $\mu(\hat{p}) = \begin{cases} \mu_b(\hat{p}) & \text{if } \hat{p} \in \mathsf{out}(S_i, C'_b) \\ \emptyset & \text{if } \hat{p} \in \mathsf{out}(S_i, C') \setminus \mathsf{out}(S_i, C'_b) \\ \bigcup_{\hat{p_o} \in N(\hat{p})} \mu(\hat{p_o}) & \text{if } \hat{p} \in \mathsf{in}_c(S_i, \curlyvee(k_a, k_n, k_b)) \\ \mu_a(\hat{p}) & \text{if } \hat{p} \in \mathsf{in}_o(S_i, \curlyvee(k_a, k_n, k_b)) \end{cases}$.

We show (i) $C' \subseteq C$, (ii) $N \colon \mathsf{in}(S_i, C') \to \wp(\mathsf{out}(S_i, C'))$, (iii) $\mu \in \overline{\mathsf{port}(S_i, C')}$, and (iv) $\forall \hat{p_i} \in \mathsf{in}(S_i, C') \colon N(\hat{p_i}) \neq \emptyset \implies \mu(\hat{p_i}) = \bigcup_{\hat{p_o} \in N(\hat{p_i})} \mu(\hat{p_o})$; to have $\curlyvee(k_a, k_n, k_b) \in \mathcal{K}(S_i)$ by Def. 7.

(i) $C' \subseteq C$: Since $C' = C'_a$ and $C'_a \subseteq C$.

(ii) $N \colon \mathsf{in}(S_i, C') \to \wp(\mathsf{out}(S_i, C'))$: We show well-definedness of N:

- Existence: Let $(c, p) \in \mathsf{in}(S_i, C')$.
  - Case $c \notin C'_n$: $N((c, p)) = \emptyset \subseteq \mathsf{out}(S_i, C')$.
  - Case $c \in C'_n$: $N((c, p)) = \{(c', p') \in N_n((c, p)) \mid c' \in C'\} \subseteq \mathsf{out}(S_i, C')$.

- Uniqueness: Let $(c,p) = (c',p') \in \mathsf{in}(S_i, C')$.
  - Case $c \notin C'_n$: First have $N((c,p)) = \emptyset$. Then, since $c' = c$ have $c' \notin C'_n$ and conclude $N((c',p')) = \emptyset$. Thus, have $N((c,p)) = \emptyset = N((c',p'))$.
  - Case $c \in C'_n$: First have $N((c,p)) = \{(c'',p'') \in N_n((c,p)) \mid c' \in C'\}$. Then, since $c' = c$ have $c' \in C'_n$ and conclude $N((c',p')) = \{(c'',p'') \in N_n((c',p')) \mid c'' \in C'\}$. Thus, since $(c,p) = (c',p')$ conclude $N_n((c,p)) = N_n((c',p'))$ and thus have $N((c,p)) = \{(c'',p'') \in N_n((c,p)) \mid c' \in C'\} = \{(c'',p'') \in N_n((c',p')) \mid c' \in C'\} = N((c',p'))$.

(iii) $\forall (c,p) \in \mathsf{in}(S_i, C'), (c_o, p_o) \in N((c,p))\colon T_{p_o} \subseteq T_p$: Let $(c,p) \in \mathsf{in}(S_i, C')$ and conclude $(c_o, p_o) \in N_n((c,p))$ by Def. 12. Thus, since $k_n \in \mathcal{K}(S_i)$ conclude $T_{p_o} \subseteq T_p$ by Def. 7.

(iv) $\mu \in \overline{\mathsf{port}(S_i, C')}$: We show well-definedness of μ:

- Existence: Let $(c,p) \in \mathsf{port}(S_i, C')$.
  - $(c,p) \in \mathsf{out}(S_i, C'_b)$: $\mu((c,p)) = \mu_b((c,p)) \subseteq T_p$.
  - $(c,p) \in \mathsf{out}(S_i, C') \setminus \mathsf{out}(S_i, C'_b)$: $\mu((c,p)) = \emptyset \subseteq T_p$.
  - $(c,p) \in \mathsf{in}_c(S_i, \curlyvee(k_a, k_n, k_b))$: $\mu((c,p)) = \bigcup_{\hat{p_o} \in N((c,p))} \mu(\hat{p_o})$ and since $\forall (c_o, p_o) \in N_n((c,p))\colon T_{p_o} \subseteq T_p$ have $\mu((c,p)) \subseteq T_p$.
  - $(c,p) \in \mathsf{in}_o(S_i, \curlyvee(k_a, k_n, k_b))$: $\mu((c,p)) = \mu_a((c,p)) \subseteq T_p$.
- Uniqueness: Let $(c,p) = (c',p') \in \mathsf{in}(S_i, C')$.
  - $(c,p) \in \mathsf{out}(S_i, C'_b)$: First have $\mu((c,p)) = \mu_b((c,p))$. Moreover, since $(c,p) = (c',p')$ have $(c',p') \in \mathsf{out}(S_i, C'_b)$ and conclude $\mu((c',p')) = \mu_b((c',p'))$. Thus, since $\mu_b((c,p)) = \mu_b((c',p'))$ conclude $\mu((c,p)) = \mu_b((c,p)) = \mu_b((c',p')) = \mu((c',p'))$.
  - $(c,p) \in \mathsf{out}(S_i, C') \setminus \mathsf{out}(S_i, C'_b)$: First have $\mu((c,p)) = \emptyset$. Moreover, since $(c,p) = (c',p')$ have $(c',p') \in \mathsf{out}(S_i, C') \setminus \mathsf{out}(S_i, C'_b)$ and conclude $\mu((c',p')) = \emptyset$. Thus, since $\mu_b((c,p)) = \mu_b((c',p'))$ conclude $\mu((c,p)) = \emptyset = \mu((c',p'))$.
  - $(c,p) \in \mathsf{in}_c(S_i, \curlyvee(k_a, k_n, k_b))$: First have $\mu((c,p)) = \bigcup_{\hat{p_o} \in N((c,p))} \mu(\hat{p_o})$. Moreover, since $(c,p) = (c',p')$ have $(c',p') \in \mathsf{in}_c(S_i, \curlyvee(k_a, k_n, k_b))$ and conclude $\mu((c',p')) = \bigcup_{\hat{p_o} \in N((c',p'))} \mu(\hat{p_o})$. Thus, since $N((c,p)) = N((c',p'))$ conclude $\mu((c,p)) = \bigcup_{\hat{p_o} \in N((c,p))} \mu(\hat{p_o}) = \bigcup_{\hat{p_o} \in N((c',p'))} \mu(\hat{p_o}) = \mu((c',p'))$.
  - $(c,p) \in \mathsf{in}_o(S_i, \curlyvee(k_a, k_n, k_b))$: First have $\mu((c,p)) = \mu_a((c,p))$. Moreover, since $(c,p) = (c',p')$ have $(c',p') \in$

$\mathsf{in}_o(S_i, \curlyvee(k_a, k_n, k_b))$ and conclude $\mu((c', p')) = \mu_a((c', p'))$. Thus, since $\mu_a((c, p)) = \mu_a((c', p'))$ conclude $\mu((c, p)) = \mu_a((c, p)) = \mu_a((c', p')) = \mu((c', p'))$.

(v) $\forall \hat{p}_i \in \mathsf{in}(S_i, C') \colon N(\hat{p}_i) \neq \emptyset \implies \mu(\hat{p}_i) = \bigcup_{\hat{p_o} \in N(\hat{p}_i)} \mu(\hat{p_o})$: Therefore, let $\hat{p}_i \in \mathsf{in}(S_i, C')$ and assume $N(\hat{p}_i) \neq \emptyset$. Thus, by Def. 10, $\hat{p}_i \in \mathsf{in}_c(S_i, \curlyvee(k_a, k_n, k_b))$ and $\mu(\hat{p}_i) = \bigcup_{\hat{p_o} \in N(\hat{p}_i)} \mu(\hat{p_o})$ by definition.

Uniqueness: Let $(C'', N', \mu') \in \mathcal{K}(S_i)$ such that $C'' = C'_a$, $N' \colon \mathsf{in}(S_i, C'') \to \wp(\mathsf{out}(S_i, C''))$, such that $N'((c,p)) = \begin{cases} \emptyset & \text{if } c \notin C'_n \\ \{(c', p') \in N_n((c,p)) \mid c' \in C''\} & \text{if } c \in C'_n \end{cases}$, and $\mu' \in \overline{\mathsf{port}(S_i, C'')}$, such that $\mu'(\hat{p}) = \begin{cases} \mu_b(\hat{p}) & \text{if } \hat{p} \in \mathsf{out}(S_i, C'_b) \\ \emptyset & \text{if } \hat{p} \in \mathsf{out}(S_i, C'') \setminus \mathsf{out}(S_i, C'_b) \\ \bigcup_{\hat{p_o} \in N'(\hat{p})} \mu'(\hat{p_o}) & \text{if } \hat{p} \in \mathsf{in}_c(S_i, \curlyvee(k_a, k_n, k_b)) \\ \mu_a(\hat{p}) & \text{if } \hat{p} \in \mathsf{in}_o(S_i, \curlyvee(k_a, k_n, k_b)) \end{cases}$.

We show $(C'', N', \mu') = (C', N, \mu)$:

- $C'' = C'$: Since $C'' = C'_a$ and $C' = C'_a$ conclude $C'' = C'$.
- $N' = N$: First note that $\mathsf{in}(S_i, C'') = \mathsf{in}(S_i, C')$ and $\mathsf{out}(S_i, C'') = \mathsf{out}(S_i, C')$. Let $(c, p) \in \mathsf{in}(S_i, C'')$.
  - Case $c \notin C'_n$: Have $N'((c,p)) = \emptyset$ and $N((c,p)) = \emptyset$ and conclude $N'((c,p)) = N((c,p))$.
  - Case $c \in C'_n$: Have $N'((c,p)) = \{(c', p') \in N_n((c,p)) \mid c' \in C''\}$ and $N((c,p)) = \{(c', p') \in N_n((c,p)) \mid c' \in C'\}$ and since $C'' = C'$ conclude $N'((c,p)) = N((c,p))$.
- $\mu' = \mu$: First note that $\mathsf{port}(S_i, C'') = \mathsf{port}(S_i, C')$ and let $(c, p) \in \mathsf{port}(S_i, C'')$.
  - $(c, p) \in \mathsf{out}(S_i, C'_b)$: Have $\mu'((c,p)) = \mu_b((c,p))$ and $\mu((c,p)) = \mu_b((c,p))$ by definition and conclude $\mu'((c,p)) = \mu((c,p))$.
  - $(c, p) \in \mathsf{out}(S_i, C'') \setminus \mathsf{out}(S_i, C'_b)$: First have $\mu'((c,p)) = \emptyset$ by definition. Moreover, since $\mathsf{out}(S_i, C') = \mathsf{out}(S_i, C'')$ have $(c, p) \in \mathsf{out}(S_i, C') \setminus \mathsf{out}(S_i, C'_b)$ and conclude $\mu((c,p)) = \emptyset$ by definition. Thus, conclude $\mu'((c,p)) = \mu((c,p))$.
  - $(c, p) \in \mathsf{in}_c(S_i, \curlyvee(k_a, k_n, k_b))$: First have $\mu'((c,p)) = \bigcup_{\hat{p_o} \in N'((c,p))} \mu'(\hat{p_o})$ and $\mu((c,p)) = \bigcup_{\hat{p_o} \in N((c,p))} \mu(\hat{p_o})$ by definition. Thus, since $N' = N$ and $\forall \hat{p} \in \mathsf{out}(S_i, C'') = \mathsf{out}(S_i, C') \colon \mu'(\hat{p}) = \mu(\hat{p})$ conclude $\mu'((c,p)) = \bigcup_{\hat{p_o} \in N'((c,p))} \mu'(\hat{p_o}) = \bigcup_{\hat{p_o} \in N((c,p))} \mu(\hat{p_o}) = \mu((c,p))$.
  - $(c, p) \in \mathsf{in}_o(S_i, \curlyvee(k_a, k_n, k_b))$: Have $\mu'((c,p)) = \mu_a((c,p))$ and

$\mu((c,p)) = \mu_a((c,p))$ by definition. Thus, conclude $\mu'((c,p)) = \mu((c,p))$. □

A.3 Proof of Lem. 2

For a function $f\colon A \to B$, in the following, we denote with $\mathrm{dom}\,(f) \stackrel{\text{def}}{=} A$ the domain of f and with $\mathrm{ran}\,(f) \stackrel{\text{def}}{=} B$ its range.

Let $(C', N, \mu) = \curlyvee\big(k_a, k_a, k_a\big)$. We show $k_a = (C', N, \mu)$.

- $C' = C'_a$: By Def. 12.
- $N = N_a$: By Def. 12, $N\colon \mathsf{in}(S_i, C') \to \wp(\mathsf{out}(S_i, C'))$ and since $C' = C'_a$ conclude $\mathrm{dom}\,(N) = \mathrm{dom}\,(N_a)$ and $\mathrm{ran}\,(N) = \mathrm{ran}\,(N_a)$. We show $\forall \hat{p} \in \mathsf{in}(S_i, C'_a)\colon N(\hat{p}) = N_a(\hat{p})$ to conclude $N = N_a$. Therefore let $(c,p) \in \mathsf{in}(S_i, C'_a)$ and conclude $c \in C'_a$. Thus, have $N((c,p)) = \{(c',p') \in N_a((c,p)) \mid c' \in C'\} = N_a((c,p))$ by Def. 12.
- $\mu = \mu_a$: By Def. 12, $\mu \in \mathsf{port}(S_i, C')$ and since $C' = C'_a$ conclude $\mathrm{dom}\,(\mu) = \mathrm{dom}\,(\mu_a)$ and $\mathrm{ran}\,(\mu) = \mathrm{ran}\,(\mu_a)$. We show $\forall \hat{p} \in \mathsf{port}(S_i, C'_a)\colon \mu(\hat{p}) = \mu_a(\hat{p})$ to conclude $\mu = \mu_a$. Therefore let $\hat{p} \in \mathsf{port}(S_i, C'_a)$.
  - Case $\hat{p} \in \mathsf{out}(S_i, C'_a)$: By Def. 12, $\mu(\hat{p}) = \mu_a(\hat{p})$.
  - Case $\hat{p} \in \mathsf{in}_c(S_i, k_a)$: Since $C' = C'_a$ and $N = N_a$ conclude $\mathsf{in}_c(S_i, \curlyvee\big(k_a, k_a, k_a\big)) = \mathsf{in}_c(S_i, k_a)$. Thus, have $\mu(\hat{p}) = \bigcup_{\hat{p_o} \in N(\hat{p})} \mu(\hat{p_o})$ by Def. 12. Thus, since $\forall \hat{p_o} \in \mathsf{out}(S_i, C'_a)\colon \mu(\hat{p_o}) = \mu_a(\hat{p_o})$ and since $N(\hat{p}) = N_a(\hat{p})$ have $\bigcup_{\hat{p_o} \in N(\hat{p})} \mu(\hat{p_o}) = \bigcup_{\hat{p_o} \in N_a(\hat{p})} \mu_a(\hat{p_o})$ and conclude $\mu(\hat{p}) = \bigcup_{\hat{p_o} \in N_a(\hat{p})} \mu_a(\hat{p_o})$. Moreover, since $\hat{p} \in \mathsf{in}_c(S_i, k_a)$ conclude $N_a(\hat{p}) \neq \emptyset$ by Def. 10. Thus, have $\mu_a(\hat{p}) = \bigcup_{\hat{p_o} \in N_a(\hat{p})} \mu_a(\hat{p_o})$ by Def. 7 and since $\mu(\hat{p}) = \bigcup_{\hat{p_o} \in N_a(\hat{p})} \mu_a(\hat{p_o})$ conclude $\mu(\hat{p}) = \mu_a(\hat{p})$.
  - Case $\hat{p} \in \mathsf{in}_o(S_i, k_a)$: Since $C' = C'_a$ and $N = N_a$ conclude $\mathsf{in}_o(S_i, \curlyvee\big(k_a, k_a, k_a\big)) = \mathsf{in}_o(S_i, k_a)$. Thus, have $\mu(\hat{p}) = \mu_a(\hat{p})$ by Def. 12. □

A.4 Proof of Lem. 3

Reflexivity: For every architecture configuration $k = (C', N, \mu) \in \mathcal{K}(S_i)$ we have $C' = C'$ and thus $k \approx^a k$ by Def. 13.

Symmetry: Let $k = (C', N, \mu) \in \mathcal{K}(S_i)$ and $k' = (C'', N', \mu') \in \mathcal{K}(S_i)$, such that $k \approx^a k'$. By Def. 13 we have $C' = C''$ and thus $k' \approx^a k$ by Def. 13 again.

Transitivity: Let $k = (C', N, \mu)$, $k' = (C'', N', \mu')$, and $k'' = (C''', N'', \mu'')$ three architecture configurations over interface specification $S_i \in \mathcal{S}_I$ such that $k \approx^a k'$ and $k' \approx^a k''$. By Def. 13 we have $C' = C''$ and $C'' = C'''$. Thus, conclude $C' = C'''$ and have $k \approx^a k''$ by Def. 13. □

A.5 Proof of Lem. 4

By Def. 12 have $\left[\curlyvee\left(k_a, k_n, k_b\right)\right]^1 = \left[k_a\right]^1$. Thus, by Def. 13 conclude $\curlyvee\left(k_a, k_n, k_b\right) \approx^a k_a$. □

A.6 Proof of Lem. 5

Reflexivity: Let $k = (C', N, \mu) \in \mathcal{K}(S_i)$. We have $\forall \hat{p_i} \in \mathsf{in}(S_i, C' \cap C') \colon N(\hat{p_i}) = N(\hat{p_i})$. Moreover, since $\mathsf{in}(S_i, C' \setminus C') = \emptyset$, we conclude $\forall \hat{p_i} \in \mathsf{in}(S_i, C' \setminus C') \colon N(\hat{p_i}) = \emptyset$. Thus, conclude $k \approx^n k$ by Def. 14.

Symmetry: Let $k = (C', N, \mu)$ and $k' = (C'', N', \mu')$ be two architecture configurations over interface specification $S_i \in \mathcal{S}_I$, such that $k \approx^n k'$. Thus, have $\forall \hat{p_i} \in \mathsf{in}(S_i, C' \cap C'') \colon N(\hat{p_i}) = N'(\hat{p_i})$, $\forall \hat{p_i} \in \mathsf{in}(S_i, C' \setminus C'') \colon N(\hat{p_i}) = \emptyset$, and $\forall \hat{p_i} \in \mathsf{in}(S_i, C'' \setminus C') \colon N'(\hat{p_i}) = \emptyset$ by Def. 14 and conclude $\forall \hat{p_i} \in \mathsf{in}(S_i, C'' \cap C') \colon N'(\hat{p_i}) = N(\hat{p_i})$, $\forall \hat{p_i} \in \mathsf{in}(S_i, C'' \setminus C') \colon N'(\hat{p_i}) = \emptyset$, and $\forall \hat{p_i} \in \mathsf{in}(S_i, C' \setminus C'') \colon N(\hat{p_i}) = \emptyset$. Thus, have $k' \approx^n k$ by Def. 14.

Transitivity: Let $k = (C', N, \mu)$, $k' = (C'', N', \mu')$, and $k'' = (C''', N'', \mu'')$ be architecture configurations over interface specification $S_i \in \mathcal{S}_I$, such that $k \approx^n k'$ and $k' \approx^n k''$. We show $k \approx^n k''$. According to Def. 14 we have to show (i) $\forall \hat{p_i} \in \mathsf{in}(S_i, C' \cap C''') \colon N(\hat{p_i}) = N''(\hat{p_i})$, (ii) $\forall \hat{p_i} \in \mathsf{in}(S_i, C' \setminus C''') \colon N(\hat{p_i}) = \emptyset$, and (iii) $\forall \hat{p_i} \in \mathsf{in}(S_i, C''' \setminus C') \colon N''(\hat{p_i}) = \emptyset$:

(i) $\forall \hat{p_i} \in \mathsf{in}(S_i, C' \cap C''') \colon N(\hat{p_i}) = N''(\hat{p_i})$: Therefore, let $\hat{p_i} \in \mathsf{in}(S_i, C' \cap C''')$.
   - Case $\hat{p_i} \notin \mathsf{in}(S_i, C'')$: Since $\hat{p_i} \in \mathsf{in}(S_i, C')$ and $k \approx^n k'$ have $N(\hat{p_i}) = \emptyset$ by Def. 14. Moreover, since $\hat{p_i} \in \mathsf{in}(S_i, C''')$ and $k' \approx^n k''$ have $N''(\hat{p_i}) = \emptyset$ by Def. 14. Thus, conclude $N(\hat{p_i}) = \emptyset = N''(\hat{p_i})$.
   - Case $\hat{p_i} \in \mathsf{in}(S_i, C'')$: Since $\hat{p_i} \in \mathsf{in}(S_i, C')$ and $k \approx^n k'$ have $N(\hat{p_i}) = N'(\hat{p_i})$ by Def. 14. Moreover, since $\hat{p_i} \in \mathsf{in}(S_i, C''')$ and $k' \approx^n k''$ have $N'(\hat{p_i}) = N''(\hat{p_i})$ by Def. 14. Thus, conclude $N(\hat{p_i}) = N'(\hat{p_i}) = N''(\hat{p_i})$.

(ii) $\forall \hat{p_i} \in \mathsf{in}(S_i, C' \setminus C''') \colon N(\hat{p_i}) = \emptyset$: Therefore, let $\hat{p_i} \in \mathsf{in}(S_i, C' \setminus C''')$.
   - Case $\hat{p_i} \notin \mathsf{in}(S_i, C'')$: Since $\hat{p_i} \in \mathsf{in}(S_i, C')$ and $k \approx^n k'$ have $N(\hat{p_i}) = \emptyset$ by Def. 14.

- Case $\hat{p}_i \in \mathsf{in}(S_i, C'')$: Since $\hat{p}_i \in \mathsf{in}(S_i, C')$ and $k \approx^n k'$ have $N(\hat{p}_i) = N'(\hat{p}_i)$ by Def. 14. Moreover, since $\hat{p}_i \notin \mathsf{in}(S_i, C''')$ and $k' \approx^n k''$ have $N'(\hat{p}_i) = \emptyset$ by Def. 14. Thus, conclude $N(\hat{p}_i) = N'(\hat{p}_i) = \emptyset$.

(iii) $\forall \hat{p}_i \in \mathsf{in}(S_i, C''' \setminus C')$: $N''(\hat{p}_i) = \emptyset$: Therefore, let $\hat{p}_i \in \mathsf{in}(S_i, C''' \setminus C')$.

- Case $\hat{p}_i \notin \mathsf{in}(S_i, C'')$: Since $\hat{p}_i \in \mathsf{in}(S_i, C''')$ and $k' \approx^n k''$ have $N''(\hat{p}_i) = \emptyset$ by Def. 14.
- Case $\hat{p}_i \in \mathsf{in}(S_i, C'')$: Since $\hat{p}_i \in \mathsf{in}(S_i, C''')$ and $k' \approx^n k''$ have $N''(\hat{p}_i) = N'(\hat{p}_i)$ by Def. 14. Moreover, since $\hat{p}_i \notin \mathsf{in}(S_i, C')$ and $k \approx^n k'$ have $N'(\hat{p}_i) = \emptyset$ by Def. 14. Thus, conclude $N''(\hat{p}_i) = N'(\hat{p}_i) = \emptyset$. □

A.7 Proof of Lem. 6

Let $(C', N, \mu) = \curlyvee\big(k_a, k_n, k_b\big)$. We show (i) $\forall \hat{p}_i \in \mathsf{in}(S_i, C' \cap C'_n)$: $N(\hat{p}_i) = N_n(\hat{p}_i)$, (ii) $\forall \hat{p}_i \in \mathsf{in}(S_i, C' \setminus C'_n)$: $N(\hat{p}_i) = \emptyset$, and (iii) $\forall \hat{p}_i \in \mathsf{in}(S_i, C'_n \setminus C')$: $N_n(\hat{p}_i) = \emptyset$; to conclude $\curlyvee\big(k_a, k_n, k_b\big) \approx^n k_n$ by Def. 14:

(i) $\forall \hat{p}_i \in \mathsf{in}(S_i, C' \cap C'_n)$: $N(\hat{p}_i) = N_n(\hat{p}_i)$: Let $(c_i, p_i) \in \mathsf{in}(S_i, C' \cap C'_n)$ and conclude $(c_i, p_i) \in \mathsf{in}(S_i, C')$ and $(c_i, p_i) \in \mathsf{in}(S_i, C'_n)$.

- $N((c_i, p_i)) \subseteq N_n((c_i, p_i))$: Let $\hat{p}_o \in N((c_i, p_i))$. By Def. 12 have $\hat{p}_o \in N_n((c_i, p_i))$.
- $N_n((c_i, p_i)) \subseteq N((c_i, p_i))$: Let $(c_o, p_o) \in N_n((c_i, p_i))$. Since $(c_i, p_i) \in \mathsf{in}(S_i, C'_n)$ and $(c_o, p_o) \in N_n((c_i, p_i))$ have $c_i \in C'_a$ and $c_o \in C'_a$ by Def. 11. Thus, since $c_i \in C'_n$ and $c_o \in C'_a$ conclude $(c_o, p_o) \in N((c_i, p_i))$ by Def. 12.

(ii) $\forall \hat{p}_i \in \mathsf{in}(S_i, C' \setminus C'_n)$: $N(\hat{p}_i) = \emptyset$: Let $(c_i, p_i) \in \mathsf{in}(S_i, C' \setminus C'_n)$ and conclude $(c_i, p_i) \notin \mathsf{in}(S_i, C'_n)$. Thus, have $c_i \notin C'_n$ and conclude $N((c_i, p_i)) = \emptyset$ by Def. 12.

(iii) $\forall \hat{p}_i \in \mathsf{in}(S_i, C'_n \setminus C')$: $N_n(\hat{p}_i) = \emptyset$: Let $(c_i, p_i) \in \mathsf{in}(S_i, C'_n \setminus C')$ and conclude $(c_i, p_i) \notin \mathsf{in}(S_i, C')$. Assume $N_n((c_i, p_i)) \neq \emptyset$ and let $(c_o, p_o) \in N_n((c_i, p_i))$. Since $(c_i, p_i) \in \mathsf{in}(S_i, C'_n)$ and $(c_o, p_o) \in N_n((c_i, p_i))$ have $c_i \in C'_a$ and $c_o \in C'_a$ by Def. 11. Thus, since $C' = C'_a$ by Def. 12 conclude $c_i \in C'$ which is in contradiction to $(c_i, p_i) \notin \mathsf{in}(S_i, C')$. □

A.8 Proof of Lem. 8

Let $(C', N, \mu) = \curlyvee\big(k_a, k_n, k_b\big)$. We show (i) $\forall \hat{p} \in \mathsf{out}(S_i, C' \cap C'_b)$: $\mu(\hat{p}) = \mu_b(\hat{p})$, (ii) $\forall \hat{p} \in \mathsf{out}(S_i, C' \setminus C'_b)$: $\mu(\hat{p}) = \emptyset$, and (iii) $\forall \hat{p} \in \mathsf{out}(S_i, C'_b \setminus C')$: $\mu_b(\hat{p}) = \emptyset$; to conclude $\curlyvee\big(k_a, k_n, k_b\big) \approx^b k_b$ by Def. 15:

(i) $\forall \hat{p} \in \mathsf{out}(S_i, C' \cap C'_b)\colon \mu(\hat{p}) = \mu_b(\hat{p})$: Let $\hat{p} \in \mathsf{out}(S_i, C' \cap C'_b)$ and conclude $\hat{p} \in \mathsf{out}(S_i, C'_b)$. Thus, have $\mu(\hat{p}) = \mu_b(\hat{p})$ by Def. 12.

(ii) $\forall \hat{p} \in \mathsf{out}(S_i, C' \setminus C'_b)\colon \mu(\hat{p}) = \emptyset$: Let $\hat{p} \in \mathsf{out}(S_i, C' \setminus C'_b)$ and conclude $\hat{p} \in \mathsf{in}(S_i, C') \setminus \mathsf{in}(S_i, C'_b)$. Thus, have $\mu(\hat{p}) = \emptyset$ by Def. 12.

(iii) $\forall \hat{p} \in \mathsf{out}(S_i, C'_b \setminus C')\colon \mu_b(\hat{p}) = \emptyset$: Let $(c, p) \in \mathsf{out}(S_i, C'_b \setminus C')$ and conclude $(c, p) \in \mathsf{in}(S_i, C'_b)$ and $(c, p) \notin \mathsf{in}(S_i, C')$. Assume $\mu_b((c, p)) \neq \emptyset$. Thus, since $(c, p) \in \mathsf{out}(S_i, C'_b)$ have $c \in C'_a$ by Def. 11. Moreover, have $C' = C'_a$ by Def. 12 and since $c \in C'_a$ conclude $c \in C'$. Thus, have a contradiction with $(c, p) \notin \mathsf{in}(S_i, C')$. □

A.9 Proof of Lem. 9

Reflexivity: Let $k = (C', N, \mu) \in \mathcal{K}(S_i)$. We have $\forall \hat{p} \in \mathsf{in}(S_i, C' \cap C')\colon \mu(\hat{p}) = \mu(\hat{p})$. Moreover, since $\mathsf{in}(S_i, C' \setminus C') = \emptyset$, we conclude $\forall \hat{p} \in \mathsf{in}(S_i, C' \setminus C')\colon \mu(\hat{p}) = \emptyset$. Thus, conclude $k \approx^b k$ by Def. 15.

Symmetry: Let $k = (C', N, \mu)$ and $k' = (C'', N', \mu')$ be two architecture configurations over interface specification $S_i \in \mathcal{S}_I$, such that $k \approx^b k'$. Thus, have $\forall \hat{p} \in \mathsf{in}(S_i, C' \cap C'')\colon \mu(\hat{p}) = \mu'(\hat{p})$, $\forall \hat{p} \in \mathsf{in}(S_i, C' \setminus C'')\colon \mu(\hat{p}) = \emptyset$, and $\forall \hat{p} \in \mathsf{in}(S_i, C'' \setminus C')\colon \mu'(\hat{p}) = \emptyset$ by Def. 15 and conclude $\forall \hat{p} \in \mathsf{in}(S_i, C'' \cap C')\colon \mu'(\hat{p}) = \mu(\hat{p})$, $\forall \hat{p} \in \mathsf{in}(S_i, C'' \setminus C')\colon \mu'(\hat{p}) = \emptyset$, and $\forall \hat{p} \in \mathsf{in}(S_i, C' \setminus C'')\colon \mu(\hat{p}) = \emptyset$. Thus, have $k' \approx^b k$ by Def. 15.

Transitivity: Let $k = (C', N, \mu)$, $k' = (C'', N', \mu')$, and $k'' = (C''', N'', \mu'')$ be architecture configurations over interface specification $S_i \in \mathcal{S}_I$, such that $k \approx^b k'$ and $k' \approx^b k''$. We show $k \approx^b k''$. According to Def. 15 we have to show $\forall \hat{p} \in \mathsf{in}(S_i, C' \cap C''')\colon \mu(\hat{p}) = \mu''(\hat{p})$, $\forall \hat{p} \in \mathsf{in}(S_i, C' \setminus C''')\colon \mu(\hat{p}) = \emptyset$, and $\forall \hat{p} \in \mathsf{in}(S_i, C''' \setminus C')\colon \mu''(\hat{p}) = \emptyset$.

- $\forall \hat{p} \in \mathsf{in}(S_i, C' \cap C''')\colon \mu(\hat{p}) = \mu''(\hat{p})$: Therefore, let $\hat{p} \in \mathsf{in}(S_i, C' \cap C''')$.
  - Case $\hat{p} \notin \mathsf{in}(S_i, C'')$: Since $\hat{p} \in \mathsf{in}(S_i, C')$ and $k \approx^b k'$ have $\mu(\hat{p}) = \emptyset$ by Def. 15. Moreover, since $\hat{p} \in \mathsf{in}(S_i, C''')$ and $k' \approx^b k''$ have $\mu''(\hat{p}) = \emptyset$ by Def. 15. Thus, conclude $\mu(\hat{p}) = \emptyset = \mu''(\hat{p})$.
  - Case $\hat{p} \in \mathsf{in}(S_i, C'')$: Since $\hat{p} \in \mathsf{in}(S_i, C')$ and $k \approx^b k'$ have $\mu(\hat{p}) = \mu'(\hat{p})$ by Def. 15. Moreover, since $\hat{p} \in \mathsf{in}(S_i, C''')$ and $k' \approx^b k''$ have $\mu'(\hat{p}) = \mu''(\hat{p})$ by Def. 15. Thus, conclude $\mu(\hat{p}) = \mu'(\hat{p}) = \mu''(\hat{p})$.
- $\forall \hat{p} \in \mathsf{in}(S_i, C' \setminus C''')\colon \mu(\hat{p}) = \emptyset$: Therefore, let $\hat{p} \in \mathsf{in}(S_i, C' \setminus C''')$.
  - Case $\hat{p} \notin \mathsf{in}(S_i, C'')$: Since $\hat{p} \in \mathsf{in}(S_i, C')$ and $k \approx^b k'$ have $\mu(\hat{p}) = \emptyset$ by Def. 15.
  - Case $\hat{p} \in \mathsf{in}(S_i, C'')$: Since $\hat{p} \in \mathsf{in}(S_i, C')$ and $k \approx^b k'$ have $\mu(\hat{p}) = \mu'(\hat{p})$ by Def. 15. Moreover, since $\hat{p} \notin \mathsf{in}(S_i, C''')$ and $k' \approx^b k''$

have $\mu'(\hat{p}) = \emptyset$ by Def. 15. Thus, conclude $\mu(\hat{p}) = \mu'(\hat{p}) = \emptyset$.

- $\forall \hat{p} \in \mathsf{in}(S_i, C''' \setminus C') \colon \mu''(\hat{p}) = \emptyset$: Therefore, let $\hat{p} \in \mathsf{in}(S_i, C''' \setminus C')$.
  - Case $\hat{p} \notin \mathsf{in}(S_i, C'')$: Since $\hat{p} \in \mathsf{in}(S_i, C''')$ and $k' \approx^b k''$ have $\mu''(\hat{p}) = \emptyset$ by Def. 15.
  - Case $\hat{p} \in \mathsf{in}(S_i, C'')$: Since $\hat{p} \in \mathsf{in}(S_i, C''')$ and $k' \approx^b k''$ have $\mu''(\hat{p}) = \mu'(\hat{p})$ by Def. 15. Moreover, since $\hat{p} \notin \mathsf{in}(S_i, C')$ and $k \approx^b k'$ have $\mu'(\hat{p}) = \emptyset$ by Def. 15. Thus, conclude $\mu''(\hat{p}) = \mu'(\hat{p}) = \emptyset$. □

A.10 Proof of Lem. 10

Let $k_a = (C'_a, N_a, \mu_a)$, $k_n = (C'_n, N_n, \mu_n)$, and $k_b = (C'_b, N_b, \mu_b)$ and assume $\curlyvee(k_a, k_n, k_b)$.

We show $k_a \approx^i k_n$: According to Def. 16 we have to show (i) $\forall \hat{p} \in \mathsf{in}(S_i, C'_a \cap C'_n) \colon \mu_a(\hat{p}) = \mu_n(\hat{p})$, (ii) $\forall \hat{p} \in \mathsf{in}(S_i, C'_a \setminus C'_n) \colon \mu_a(\hat{p}) = \emptyset$, and (iii) $\forall \hat{p} \in \mathsf{in}(S_i, C'_n \setminus C'_a) \colon \mu_n(\hat{p}) = \emptyset$.

(i) $\forall \hat{p} \in \mathsf{in}(S_i, C'_a \cap C'_n) \colon \mu_a(\hat{p}) = \mu_n(\hat{p})$: Let $\hat{p} \in \mathsf{in}(S_i, C'_a \cap C'_n)$. According to Def. 11 we have $\big(\hat{p} \notin \mathsf{in}(S_i, C'_a) \vee \mu_a(\hat{p}) = \emptyset\big) \wedge \big(\hat{p} \notin \mathsf{in}(S_i, C'_n) \vee \mu_n(\hat{p}) = \emptyset\big) \wedge \big(\hat{p} \notin \mathsf{in}(S_i, C'_b) \vee \mu_b(\hat{p}) = \emptyset\big)$ or $\hat{p} \in \mathsf{in}(S_i, C'_a) \wedge \hat{p} \in \mathsf{in}(S_i, C'_n) \wedge \hat{p} \in \mathsf{in}(S_i, C'_b) \wedge \mu_a(\hat{p}) = \mu_n(\hat{p}) = \mu_b(\hat{p})$.

- Case $\big(\hat{p} \notin \mathsf{in}(S_i, C'_a) \vee \mu_a(\hat{p}) = \emptyset\big) \wedge \big(\hat{p} \notin \mathsf{in}(S_i, C'_n) \vee \mu_n(\hat{p}) = \emptyset\big) \wedge \big(\hat{p} \notin \mathsf{in}(S_i, C'_b) \vee \mu_b(\hat{p}) = \emptyset\big)$: Since $\hat{p} \in \mathsf{in}(S_i, C'_a \cap C'_n)$ have $\hat{p} \in \mathsf{in}(S_i, C'_a)$ and since $\hat{p} \notin \mathsf{in}(S_i, C'_a) \vee \mu_a(\hat{p}) = \emptyset$ by case assumption, conclude $\mu_a(\hat{p}) = \emptyset$. Moreover, since $\hat{p} \in \mathsf{in}(S_i, C'_a \cap C'_n)$ have $\hat{p} \in \mathsf{in}(S_i, C'_n)$ and since $\hat{p} \notin \mathsf{in}(S_i, C'_n) \vee \mu_n(\hat{p}) = \emptyset$ by case assumption, conclude $\mu_n(\hat{p}) = \emptyset$. Thus, have $\mu_a(\hat{p}) = \emptyset = \mu_n(\hat{p})$.
- Case $\hat{p} \in \mathsf{in}(S_i, C'_a) \wedge \hat{p} \in \mathsf{in}(S_i, C'_n) \wedge \hat{p} \in \mathsf{in}(S_i, C'_b) \wedge \mu_a(\hat{p}) = \mu_n(\hat{p}) = \mu_b(\hat{p})$: Conclude $\mu_a(\hat{p}) = \mu_n(\hat{p})$ from case assumption.

(ii) $\forall \hat{p} \in \mathsf{in}(S_i, C'_a \setminus C'_n) \colon \mu_a(\hat{p}) = \emptyset$: Let $\hat{p} \in \mathsf{in}(S_i, C'_a \setminus C'_n)$. According to Def. 11 we have $\big(\hat{p} \notin \mathsf{in}(S_i, C'_a) \vee \mu_a(\hat{p}) = \emptyset\big) \wedge \big(\hat{p} \notin \mathsf{in}(S_i, C'_n) \vee \mu_n(\hat{p}) = \emptyset\big) \wedge \big(\hat{p} \notin \mathsf{in}(S_i, C'_b) \vee \mu_b(\hat{p}) = \emptyset\big)$ or $\hat{p} \in \mathsf{in}(S_i, C'_a) \wedge \hat{p} \in \mathsf{in}(S_i, C'_n) \wedge \hat{p} \in \mathsf{in}(S_i, C'_b) \wedge \mu_a(\hat{p}) = \mu_n(\hat{p}) = \mu_b(\hat{p})$. Moreover, since $\hat{p} \in \mathsf{in}(S_i, C'_a \setminus C'_n)$ have $\hat{p} \in \mathsf{in}(S_i, C'_a)$ and conclude $\big(\hat{p} \notin \mathsf{in}(S_i, C'_a) \vee \mu_a(\hat{p}) = \emptyset\big) \wedge \big(\hat{p} \notin \mathsf{in}(S_i, C'_n) \vee \mu_n(\hat{p}) = \emptyset\big) \wedge \big(\hat{p} \notin \mathsf{in}(S_i, C'_b) \vee \mu_b(\hat{p}) = \emptyset\big)$. Thus, since $\hat{p} \in \mathsf{in}(S_i, C'_a)$ conclude $\mu_a(\hat{p}) = \emptyset$.

(iii) $\forall \hat{p} \in \mathsf{in}(S_i, C'_n \setminus C'_a) \colon \mu_n(\hat{p}) = \emptyset$: Let $\hat{p} \in \mathsf{in}(S_i, C'_n \setminus C'_a)$. According to Def. 11 we have $\big(\hat{p} \notin \mathsf{in}(S_i, C'_a) \vee \mu_a(\hat{p}) = \emptyset\big) \wedge \big(\hat{p} \notin \mathsf{in}(S_i, C'_n) \vee \mu_n(\hat{p}) = \emptyset\big) \wedge \big(\hat{p} \notin \mathsf{in}(S_i, C'_b) \vee \mu_b(\hat{p}) = \emptyset\big)$ or $\hat{p} \in \mathsf{in}(S_i, C'_a) \wedge \hat{p} \in \mathsf{in}(S_i, C'_n) \wedge \hat{p} \in \mathsf{in}(S_i, C'_b) \wedge \mu_a(\hat{p}) = \mu_n(\hat{p}) = \mu_b(\hat{p})$. Moreover, since $\hat{p} \in \mathsf{in}(S_i, C'_n \setminus C'_a)$

have $\hat{p} \in \mathsf{in}(S_i, C'_n)$ and conclude $\big(\hat{p} \notin \mathsf{in}(S_i, C'_a) \vee \mu_a(\hat{p}) = \emptyset\big) \wedge \big(\hat{p} \notin \mathsf{in}(S_i, C'_n) \vee \mu_n(\hat{p}) = \emptyset\big) \wedge \big(\hat{p} \notin \mathsf{in}(S_i, C'_b) \vee \mu_b(\hat{p}) = \emptyset\big)$. Thus, since $\hat{p} \in \mathsf{in}(S_i, C'_n)$ conclude $\mu_n(\hat{p}) = \emptyset$.

A similar argument can be used to show $k_n \approx^i k_b$.

Finally, since $k_a \approx^i k_n$ and $k_n \approx^i k_b$ conclude $k_a \approx^i k_b$ by Lem. 9. □

A.11 Proof of Lem. 11

Let $k_a = (C'_a, N_a, \mu_a)$, $k_n = (C'_n, N_n, \mu_n)$, and $k_b = (C'_b, N_b, \mu_b)$ and $\curlyvee\big(k, k_a, k_n\big) = (C', N, \mu)$.

We show $k_a \approx^i \curlyvee\big(k_a, k_n, k_b\big)$: By Def. 16 we have to show $\forall \hat{p} \in \mathsf{in}(S_i, C'_a \cap C')\colon \mu_a(\hat{p}) = \mu(\hat{p})$, $\forall \hat{p} \in \mathsf{in}(S_i, C'_a \setminus C')\colon \mu_a(\hat{p}) = \emptyset$, and $\forall \hat{p} \in \mathsf{in}(S_i, C' \setminus C'_a)\colon \mu(\hat{p}) = \emptyset$.

- Show $\forall \hat{p} \in \mathsf{in}(S_i, C'_a \cap C')\colon \mu_a(\hat{p}) = \mu(\hat{p})$: Therefore, let $\hat{p} \in \mathsf{in}(S_i, C'_a \cap C')$ and conclude $\hat{p} \in \mathsf{in}(S_i, C'_a)$ and $\hat{p} \in \mathsf{in}(S_i, C')$.
  - Case $\hat{p} \in \mathsf{in}_o(S_i, \curlyvee\big(k_a, k_n, k_b\big))$: Conclude $\mu(\hat{p}) = \mu_a(\hat{p})$ by Def. 12.
  - Case $\hat{p} \in \mathsf{in}_c(S_i, \curlyvee\big(k_a, k_n, k_b\big))$: By Def. 10 have $N(\hat{p}) \neq \emptyset$ and conclude $\hat{p} \in \mathsf{in}(S_i, C'_n)$ and $N_n(\hat{p}) \cap \mathsf{out}(S_i, C'_a) \neq \emptyset$ by Def. 12. Thus, since $\hat{p} \in \mathsf{in}(S_i, C'_a) \cap \mathsf{in}(S_i, C'_n)$ and $N_n(\hat{p}) \cap \mathsf{out}(S_i, C'_a) \neq \emptyset$ have $\mu_a(\hat{p}) = \bigcup_{\hat{p_o} \in N_n(\hat{p}) \cap \mathsf{out}(S_i, C'_a) \cap \mathsf{out}(S_i, C'_b)} \mu_b(\hat{p_o})$ by Def. 11. Moreover, have $N_n(\hat{p}) \cap \mathsf{out}(S_i, C'_a) = N(\hat{p})$ by Def. 12 and $\forall \hat{p} \in \mathsf{out}(S_i, C'_b)\colon \mu(\hat{p}) = \mu_b(\hat{p})$ by Def. 12. Thus, conclude $\bigcup_{\hat{p_o} \in N_n(\hat{p}) \cap \mathsf{out}(S_i, C'_a) \cap \mathsf{out}(S_i, C'_b)} \mu_b(\hat{p_o}) = \bigcup_{\hat{p_o} \in N(\hat{p}) \cap \mathsf{out}(S_i, C'_b)} \mu(\hat{p_o})$. Moreover, have $\forall \hat{p} \in \mathsf{out}(S_i, C'_a) \setminus \mathsf{out}(S_i, C'_b)\colon \mu(\hat{p}) = \emptyset$ by Def. 12 and conclude $\bigcup_{\hat{p_o} \in N(\hat{p}) \cap \mathsf{out}(S_i, C'_b)} \mu(\hat{p_o}) = \bigcup_{\hat{p_o} \in N(\hat{p})} \mu(\hat{p_o})$. Thus, have $\mu_a(\hat{p}) = \bigcup_{\hat{p_o} \in N_n(\hat{p}) \cap \mathsf{out}(S_i, C'_a) \cap \mathsf{out}(S_i, C'_b)} \mu_b(\hat{p_o}) = \bigcup_{\hat{p_o} \in N(\hat{p})} \mu(\hat{p_o})$ and since $\mu(\hat{p}) = \bigcup_{\hat{p_o} \in N(\hat{p})} \mu(\hat{p_o})$ by Def. 12, conclude $\mu_a(\hat{p}) = \mu(\hat{p})$.
- Show $\forall \hat{p} \in \mathsf{in}(S_i, C'_a \setminus C')\colon \mu_a(\hat{p}) = \emptyset$: First, have $C' = C'_a$ by Def. 12 and conclude $C'_a \setminus C' = \emptyset$. Thus, have $\forall \hat{p} \in \mathsf{in}(S_i, C'_a \setminus C')\colon \mu_a(\hat{p}) = \emptyset$.
- Show $\forall \hat{p} \in \mathsf{in}(S_i, C' \setminus C'_a)\colon \mu(\hat{p}) = \emptyset$: First, have $C' = C'_a$ by Def. 12 and conclude $C'_a \setminus C' = \emptyset$. Thus, have $\forall \hat{p} \in \mathsf{in}(S_i, C' \setminus C'_a)\colon \mu(\hat{p}) = \emptyset$.

We show $k_n \approx^i \curlyvee\big(k_a, k_n, k_b\big)$: From Lem. 10 have $k_n \approx^i k_a$ and since $k_a \approx^i \curlyvee\big(k_a, k_n, k_b\big)$ conclude $k_n \approx^i \curlyvee\big(k_a, k_n, k_b\big)$ by Lem. 9.

We show $k_b \approx^i \curlyvee\big(k_a, k_n, k_b\big)$: From Lem. 10 have $k_b \approx^i k_a$ and since $k_a \approx^i \curlyvee\big(k_a, k_n, k_b\big)$ conclude $k_b \approx^i \curlyvee\big(k_a, k_n, k_b\big)$ by Lem. 9. □

A.12 Proof of Lem. 12

Let $k = (C', N, \mu)$ and $k' = (C'', N', \mu')$ two architecture configurations over interface specification $S_i \in \mathcal{S}_I$.

$\Longrightarrow$: Assume $k = k'$. We show (i) $k \approx^a k'$, (ii) $k \approx^n k'$, (iii) $k \approx^b k'$, and (iv) $k \approx^i k'$:

(i) $k \approx^a k'$: Since $C' = C''$ have $k \approx^a k'$ by Def. 13.

(ii) $k \approx^n k'$: Since $N = N'$ conclude $\forall \hat{p_i} \in \mathsf{in}(S_i, C' \cap C'')\colon N(\hat{p_i}) = N'(\hat{p_i})$. Moreover, since $C' = C''$ have $C' \setminus C'' = \emptyset$ and $C'' \setminus C' = \emptyset$. Thus, conclude $\forall \hat{p_i} \in \mathsf{in}(S_i, C' \setminus C'')\colon N(\hat{p_i}) = \emptyset$ and $\forall \hat{p_i} \in \mathsf{in}(S_i, C'' \setminus C')\colon N(\hat{p_i}) = \emptyset$. Finally have $k \approx^n k'$ by Def. 14.

(iii) $k \approx^b k'$: Since $\mu = \mu'$ conclude $\forall \hat{p} \in \mathsf{out}(S_i, C' \cap C'')\colon \mu(\hat{p}) = \mu'(\hat{p})$. Moreover, since $C' = C''$ have $C' \setminus C'' = \emptyset$ and $C'' \setminus C' = \emptyset$. Thus, conclude $\forall \hat{p} \in \mathsf{out}(S_i, C' \setminus C'')\colon \mu(\hat{p}) = \emptyset$ and $\forall \hat{p} \in \mathsf{out}(S_i, C'' \setminus C')\colon \mu'(\hat{p}) = \emptyset$. Finally have $k \approx^b k'$ by Def. 15.

(iv) $k \approx^i k'$: Since $\mu = \mu'$ conclude $\forall \hat{p} \in \mathsf{in}(S_i, C' \cap C'')\colon \mu(\hat{p}) = \mu'(\hat{p})$. Moreover, since $C' = C''$ have $C' \setminus C'' = \emptyset$ and $C'' \setminus C' = \emptyset$. Thus, conclude $\forall \hat{p} \in \mathsf{in}(S_i, C' \setminus C'')\colon \mu(\hat{p}) = \emptyset$ and $\forall \hat{p} \in \mathsf{in}(S_i, C'' \setminus C')\colon \mu'(\hat{p}) = \emptyset$. Finally have $k \approx^i k'$ by Def. 16.

$\Longleftarrow$: Assume $k \approx^a k' \wedge k \approx^n k' \wedge k \approx^b k' \wedge k \approx^i k'$. We show (i) $C' = C''$, (ii) $N = N'$, and (iii) $\mu = \mu'$; to have $k = k'$:

(i) $C' = C''$: Since $k \approx^a k'$ have $C' = C''$ by Def. 13.

(ii) $N = N'$: From $C' = C''$ have $\mathrm{dom}\,(N) = \mathsf{in}(S_i, C') = \mathsf{in}(S_i, C'') = \mathrm{dom}\,(N')$ and $\mathrm{ran}\,(N) = \wp(\mathsf{out}(C', S_i)) = \wp(\mathsf{out}(C'', S_i)) = \mathrm{ran}\,(N')$. We show that $\forall \hat{p} \in \mathsf{in}(S_i, C') \cap \mathsf{in}(S_i, C'')\colon N(\hat{p}) = N'(\hat{p})$: Therefore, let $\hat{p} \in \mathsf{in}(S_i, C' \cap C'')$ and since $k \approx^n k'$ have $N(\hat{p}) = N'(\hat{p})$ by Def. 14.

(iii) $\mu = \mu'$: From $C' = C''$ have $\mathrm{dom}\,(\mu) = \mathsf{port}(C', S_i) = \mathsf{port}(C'', S_i) = \mathrm{dom}\,(\mu')$ and $\mathrm{ran}\,(\mu) = \wp(\mathsf{M}) = \mathrm{ran}\,(\mu')$. We show that $\forall \hat{p} \in \mathsf{port}(S_i, C') \cap \mathsf{port}(S_i, C'')\colon \mu(\hat{p}) = \mu'(\hat{p})$: Therefore, let $\hat{p} \in \mathsf{port}(S_i, C' \cap C'')$ and since $k \approx^b k'$ have $\mu(\hat{p}) = \mu'(\hat{p})$ by Def. 15.□

A.13 Proof of Lem. 14

By Def. 19 we have to show that for all $t_a \in \mathsf{close}^{\mathsf{a}}(A)$ and $t_n, t_b \in \mathcal{K}^t(S_i)$, such that $\curlyvee(t_a, t_n, t_b)$, there exists a $t'_a \in \mathsf{close}^{\mathsf{a}}(A)$, such that $t'_a \approx^a t_a$, $t'_a \approx^n t_n$, $t'_a \approx^b t_b$, and $t'_a \approx^i \curlyvee\big(t_a, t_n, t_b\big)$. Therefore, let $t_a \in \mathsf{close}^{\mathsf{a}}(A)$ and $t_n, t_b \in \mathcal{K}^t(S_i)$, such that $\curlyvee(t_a, t_n, t_b)$, and conclude that for all $n \in \mathbb{N}$, $\curlyvee(t_a(n), t_n(n), t_b(n))$ by Def. 18. Then, construct $t'_a \in \mathcal{K}^t(S_i)$, such that for all $n \in \mathbb{N}$, $t'_a(n) \stackrel{\text{def}}{=} \curlyvee\big(t_a(n), t_n(n), t_b(n)\big)$. First, note that t'_a is indeed

a valid configuration trace: Since $\forall n \in \mathbb{N}\colon \curlyvee\,(t_a(n), t_n(n), t_b(n))$, conclude that $\curlyvee\big(t_a(n), t_n(n), t_b(n)\big)$ is a well-defined architecture configuration for every $n \in \mathbb{N}$ by Def. 12.

We now show $t'_a \in \mathsf{close}^{\mathsf{a}}(A)$: Since $t_a \in \mathsf{close}^{\mathsf{a}}(A)$ have $\exists t''_a \in A\colon t''_a \approx^a t_a \wedge t''_a \approx^i t_a$ by Def. 23 and conclude that for all $n \in \mathbb{N}$, $t''_a(n) \approx^a t_a(n)$ and $t''_a(n) \approx^i t_a(n)$ by Def. 18. Since $t''_a \in A$, we show (i) $t'_a \approx^a t''_a$ and (ii) $t'_a \approx^i t''_a$ to conclude $t'_a \in \mathsf{close}^{\mathsf{a}}(A)$ by Def. 23.

(i) $t'_a \approx^a t''_a$: From construction have $\forall n \in \mathbb{N}\colon t'_a(n) \approx^a t_a(n)$ by Lem. 4. Thus, since $\forall n \in \mathbb{N}\colon t''_a(n) \approx^a t_a(n)$ have $\forall n \in \mathbb{N}\colon t'_a(n) \approx^a t''_a(n)$ by Lem. 3 and conclude $t'_a \approx^a t''_a$ from Def. 18.

(ii) $t'_a \approx^i t''_a$: From construction have $\forall n \in \mathbb{N}\colon t'_a(n) \approx^i t_a(n)$ by Lem. 11. Thus, since $\forall n \in \mathbb{N}\colon t''_a(n) \approx^i t_a(n)$ have $\forall n \in \mathbb{N}\colon t'_a(n) \approx^i t''_a(n)$ by Lem. 9 and conclude $t'_a \approx^i t''_a$ from Def. 18.

We conclude by showing (i) $t'_a \approx^a t_a$, (ii) $t'_a \approx^n t_n$, (iii) $t'_a \approx^b t_b$, and (iv) $t'_a \approx^i \curlyvee\big(t_a, t_n, t_b\big)$:

(i) $t'_a \approx^a t_a$: From construction have $\forall n \in \mathbb{N}\colon t'_a(n) \approx^a t_a(n)$ by Lem. 4 and conclude $t'_a \approx^a t_n$ by Def. 18.

(ii) $t'_a \approx^n t_n$: From construction have $\forall n \in \mathbb{N}\colon t'_a(n) \approx^n t_n(n)$ by Lem. 6 and conclude $t'_a \approx^n t_n$ by Def. 18.

(iii) $t'_a \approx^b t_b$: From construction have $\forall n \in \mathbb{N}\colon t'_a(n) \approx^b t_b(n)$ by Lem. 8 and conclude $t'_a \approx^b t_b$ by Def. 18.

(iv) $t'_a \approx^i \curlyvee\big(t_a, t_n, t_b\big)$: From construction have $\forall n \in \mathbb{N}\colon t'_a(n) \approx^i \curlyvee\big(t_a(n), t_n(n), t_b(n)\big)$ by Lem. 11 and conclude $t'_a \approx^i \curlyvee\big(t_a, t_n, t_b\big)$ by Def. 18. □

A.14 Proof of Thm. 20

Let $A \stackrel{\text{def}}{=} \mathsf{close}^{\mathsf{a}}(S)$, $N \stackrel{\text{def}}{=} \mathsf{close}^{\mathsf{n}}(S)$, and $B \stackrel{\text{def}}{=} \mathsf{close}^{\mathsf{b}}(S)$ and note that A is a valid AP, N a valid CP, and B a valid BP by Lem. 14.

We now show $A \cap N \cap B = S$.

- $A \cap N \cap B \subseteq S$: Therefore, assume $t \in A \cap N \cap B$: Since $t \in A$, have $t \in \mathsf{close}^{\mathsf{a}}(S)$ from construction and conclude $\exists t_a \in S\colon t_a \approx^a t \wedge t_a \approx^i t$ by Def. 23. Moreover, since $t \in N$, have $t \in \mathsf{close}^{\mathsf{n}}(S)$ from construction and conclude $\exists t_n \in S\colon t_n \approx^n t \wedge t_n \approx^i t$ by Def. 23. Moreover, since $t \in B$, have $t \in \mathsf{close}^{\mathsf{b}}(S)$ from construction and conclude $\exists t_b \in S\colon t_b \approx^b t \wedge t_b \approx^i t$ by Def. 23. Therefore, since S is a SAP, conclude $t \in S$ by Def. 22.
- $A \cap N \cap B \supseteq S$: Therefore, assume $t \in S$. From Lem. 3 have $t \approx^a t$ and from Lem. 9 have $t \approx^i t$. Thus, since $t \in S$, have $t \in \mathsf{close}^{\mathsf{a}}(S)$ by

Def. 23 and conclude $t \in A$ by construction. Moreover, have $t \approx^n t$ by Lem. 5 and $t \approx^i t$ by Lem. 9. Thus, since $t \in S$, have $t \in \mathsf{close}^{\mathsf{n}}(S)$ by Def. 23 and conclude $t \in N$ by construction. Finally, have $t \approx^b t$ by Lem. 7 and $t \approx^i t$ by Lem. 9. Thus, since $t \in S$, have $t \in \mathsf{close}^{\mathsf{b}}(S)$ by Def. 23 and conclude $t \in B$ by construction. □

A.15 Proof of Lem. 13

Existence: $\exists t \in \mathcal{K}^t(S_i) \colon t \approx^a t_a \wedge t \approx^n t_n \wedge t \approx^b t_b \wedge t \approx^i \curlyvee\big(t_a, t_n, t_b\big)$. Therefore, let $t \in \mathcal{K}^t(S_i)$, such that for all $n \in \mathbb{N}$: $t(n) \stackrel{\text{def}}{=} \curlyvee\big(t_a(n), t_n(n), t_b(n)\big)$. Note that t is well-defined: Since $\forall n \in \mathbb{N} \colon \curlyvee(t_a(n), t_n(n), t_b(n))$, conclude that $\curlyvee\big(t_a(n), t_n(n), t_b(n)\big)$ is a well-defined architecture configuration for every $n \in \mathbb{N}$ by Def. 12. We show (i) $t \approx^a t_a$, (ii) $t \approx^n t_n$, (iii) $t \approx^b t_b$, and (iv) $t \approx^i \curlyvee\big(t_a, t_n, t_b\big)$:

(i) $t \approx^a t_a$: From construction have $\forall n \in \mathbb{N} \colon t(n) \approx^a t_a(n)$ by Lem. 4 and conclude $t \approx^a t_a$ by Def. 18.

(ii) $t \approx^n t_n$: From construction have $\forall n \in \mathbb{N} \colon t(n) \approx^n t_n(n)$ by Lem. 6 and conclude $t \approx^n t_n$ by Def. 18.

(iii) $t \approx^b t_b$: From construction have $\forall n \in \mathbb{N} \colon t(n) \approx^b t_b(n)$ by Lem. 8 and conclude $t \approx^b t_b$ by Def. 18.

(iv) $t \approx^i \curlyvee\big(t_a, t_n, t_b\big)$: From construction have $t(n) \approx^i \curlyvee\big(t_a(n), t_n(n), t_b(n)\big)$ by Lem. 9 and conclude $t \approx^i \curlyvee\big(t_a, t_n, t_b\big)$ by Def. 18.

Uniqueness: $\forall t' \in \mathcal{K}^t(S_i) \colon t \approx^a t_a \wedge t \approx^n t_n \wedge t \approx^b t_b \wedge t \approx^i \curlyvee\big(t_a, t_n, t_b\big) \implies t' = t$. Therefore, let $t' \in \mathcal{K}^t(S_i)$ and assume $t' \approx^a t_a$, $t' \approx^n t_n$, $t' \approx^b t_b$, and $t' \approx^i \curlyvee\big(t_a, t_n, t_b\big)$. Then have $\forall n \in \mathbb{N} \colon t'(n) \approx^a t_a(n) \wedge t'(n) \approx^n t_n(n) \wedge t'(n) \approx^b t_b(n) \wedge t'(n) \approx^i \curlyvee\big(t_a(n), t_n(n), t_b(n)\big)$ by Def. 18. We show that for all $n \in \mathbb{N}$ we have (i) $t'(n) \approx^a t(n)$, (ii) $t'(n) \approx^n t(n)$, (iii) $t'(n) \approx^b t(n)$, and (iv) $t'(n) \approx^i t(n)$ to have $\forall n \in \mathbb{N} \colon t'(n) = t(n)$ by Lem. 12 and conclude $t' = t$ by Def. 18.

(i) $t'(n) \approx^a t(n)$: Since $\forall n \in \mathbb{N} \colon t_a(n) \approx^a t(n)$ and since $\forall n \in \mathbb{N} \colon t'(n) \approx^a t_a(n)$ conclude $\forall n \in \mathbb{N} \colon t'(n) \approx^a t(n)$ by Lem. 3.

(ii) $t'(n) \approx^n t(n)$: Since $\forall n \in \mathbb{N} \colon t_n(n) \approx^n t(n)$ and since $\forall n \in \mathbb{N} \colon t'(n) \approx^n t_n(n)$ conclude $\forall n \in \mathbb{N} \colon t'(n) \approx^n t(n)$ by Lem. 5.

(iii) $t'(n) \approx^b t(n)$: Since $\forall n \in \mathbb{N} \colon t_b(n) \approx^b t(n)$ and since $\forall n \in \mathbb{N} \colon t'(n) \approx^b t_b(n)$ conclude $\forall n \in \mathbb{N} \colon t'(n) \approx^b t(n)$ by Lem. 7.

(iv) $t'(n) \approx^i t(n)$: Since $\forall n \in \mathbb{N} \colon \curlyvee\big(t_a(n), t_n(n), t_b(n)\big) \approx^i t(n)$ and since $\forall n \in \mathbb{N} \colon t'(n) \approx^i \curlyvee\big(t_a(n), t_n(n), t_b(n)\big)$ conclude $\forall n \in \mathbb{N} \colon t'(n) \approx^i t(n)$

by Lem. 9. □

A.16 Proof of Thm. 17

$\Longrightarrow$: Assume $t \in A \cap N \cap B$. Then, let $t_a = t_n = t_b = t$ and show $t_a \in A \wedge t_n \in N \wedge t_b \in B$, $\curlyvee(t_a, t_n, t_b)$, and $t \approx^i \curlyvee\big(t_a, t_n, t_b\big)$. We show (i) $t_a \in A$, (ii) $t_n \in N$, (iii) $t_b \in B$, (iv) $\curlyvee(t_a, t_n, t_b)$, (v) $t \approx^a t_a$, (vi) $t \approx^n t_n$, (vii) $t \approx^b t_b$, and (viii) $t \approx^i \curlyvee\big(t_a, t_n, t_b\big)$:

(i) $t_a \in A$: Since $t \in A$ and $t_a = t$, conclude $t_a \in A$.
(ii) $t_n \in N$: Since $t \in N$ and $t_n = t$, conclude $t_n \in N$.
(iii) $t_b \in B$: Since $t \in B$ and $t_b = t$, conclude $t_b \in B$.
(iv) $\curlyvee(t_a, t_n, t_b)$: Since $t_a = t_n = t_b$ have $\curlyvee(t_a, t_n, t_b)$ by Lem. 1.
(v) $t \approx^a t_a$: Since $t = t_a$ conclude $t \approx^a t_a$ by Lem. 3.
(vi) $t \approx^n t_n$: Since $t = t_n$ conclude $t \approx^n t_n$ by Lem. 5.
(vii) $t \approx^b t_b$: Since $t = t_b$ conclude $t \approx^b t_b$ by Lem. 7.
(viii) $t \approx^i \curlyvee\big(t_a, t_n, t_b\big)$: Since $t = t_a = t_n = t_b$ have $t = \curlyvee\big(t_a, t_n, t_b\big)$ from Lem. 2 and conclude $\forall n \in \mathbb{N}$: $t(n) = \curlyvee\big(t_a(n), t_n(n), t_b(n)\big)$. Thus, have $\forall n \in \mathbb{N}$: $t(n) \approx^i \curlyvee\big(t_a(n), t_n(n), t_b(n)\big)$ by Lem. 9 and conclude $t \approx^i \curlyvee\big(t_a, t_n, t_b\big)$ by Def. 18.

$\Longleftarrow$: Let $t_a \in A$, $t_n \in N$, $t_b \in B$, and assume $\curlyvee(t_a, t_n, t_b)$, $t \approx^a t_a$, $t \approx^n t_n$, $t \approx^b t_b$, and $t \approx^i \curlyvee\big(t_a, t_n, t_b\big)$. We show $t \in A \cap N \cap B$.

- $t \in A$: Since $t_a \in A$, $t_n, t_b \in \mathcal{K}^t(S_i)$ and $\curlyvee(t_a, t_n, t_b)$, conclude $\exists t'_a \in A$, such that $t'_a \approx^a t_a$, $t'_a \approx^n t_n$, $t'_a \approx^b t_b$, and $t'_a \approx^i \curlyvee\big(t_a, t_n, t_b\big)$ by Def. 19. Thus, since $t \approx^a t_a$, $t \approx^n t_n$, $t \approx^b t_b$, and $t \approx^i \curlyvee\big(t_a, t_n, t_b\big)$, have $t'_a = t$ from Lem. 13 and since $t'_a \in A$ conclude $t \in A$.
- $t \in N$: Since $t_n \in N$, $t_a, t_b \in \mathcal{K}^t(S_i)$ and $\curlyvee(t_a, t_n, t_b)$, conclude $\exists t'_n \in N$, such that $t'_n \approx^n t_n$, $t'_n \approx^a t_a$, $t'_n \approx^b t_b$, and $t'_n \approx^i \curlyvee\big(t_a, t_n, t_b\big)$ by Def. 20. Thus, since $t \approx^a t_a$, $t \approx^n t_n$, $t \approx^b t_b$, and $t \approx^i \curlyvee\big(t_a, t_n, t_b\big)$, have $t'_n = t$ from Lem. 13 and since $t'_n \in N$ conclude $t \in N$.
- $t \in B$: Since $t_b \in B$, $t_a, t_n \in \mathcal{K}^t(S_i)$ and $\curlyvee(t_a, t_n, t_b)$, conclude $\exists t'_b \in B$, such that $t'_b \approx^b t_b$, $t'_b \approx^a t_a$, $t'_b \approx^n t_n$, and $t'_b \approx^i \curlyvee\big(t_a, t_n, t_b\big)$ by Def. 21. Thus, since $t \approx^a t_a$, $t \approx^n t_n$, $t \approx^b t_b$, and $t \approx^i \curlyvee\big(t_a, t_n, t_b\big)$, have $t'_b = t$ from Lem. 13 and since $t'_b \in B$ conclude $t \in B$. □