Towards a Calculus for Dynamic Architectures

Diego Marmsoler(✉)

Technische Universität München, Munich, Germany
diego.marmsoler@tum.de

Abstract. The architecture of a system describes the system's overall organization into components and connections between those components. With the emergence of mobile computing, dynamic architectures have become increasingly important. In such architectures, components may appear or disappear, and connections may change over time. The dynamic nature of such architectures makes reasoning about their behavior difficult. Since components can be activated and deactivated over time, their behavioral specifications depend on their state of activation. To address this problem, we introduce a calculus for dynamic architectures in a natural deduction style. Therefore, we provide introduction and elimination rules for several operators traditionally employed to specify component behavior. Finally, we show *soundness* and *relative completeness* of these rules. The calculus can be used to reason about component behavior in a dynamic environment. This is demonstrated by applying it to verify a property of dynamic blackboard architectures.

Keywords: Dynamic architectures · Component calculus · Architecture verification · Configuration traces · Behavior traces

1 Introduction

A system's architecture provides a set of components and connections between their ports. With the emergence of mobile computing, dynamic architectures have become more and more important [2,8,16]. In such architectures, components can appear and disappear, and connections can change, both over time. Dynamic architectures can be modeled in terms of configuration traces [14,15]. Consider, for example, the execution trace of a dynamic architecture depicted in Fig. 1. The figure shows the first three configurations of one possible execution of a dynamic architecture composed of three components c_1, c_2, and c_3. To facilitate the specification of such architectures, they can be separated into behavioral specifications for components, activation specifications, and connection specifications [15]. Thereby, behavior of components is often specified by means of temporal logic formulæ [13] over the components interface. Consider, for example, a component c_3 with output port o_1 whose behavior is given by the temporal specification "$\bigcirc(o_1 = 8)$", meaning that it outputs an 8 on its port o_1 at time point 1 (assuming that time starts at 0).

D.V. Hung and D. Kapur (Eds.): ICTAC 2017, LNCS 10580, pp. 79–99, 2017.
DOI: 10.1007/978-3-319-67729-3_6

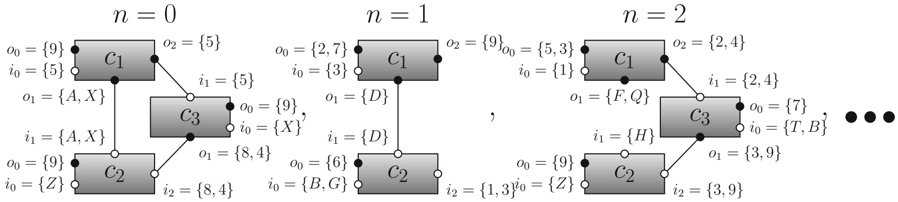

Fig. 1. Execution trace of a dynamic architecture.

For *static* architectures, the original specification of temporal properties of single components remain valid even when deployed to the architecture. The original specification of component c_3, for example, is still valid when deployed to a static architecture, i.e., c_3 will still output an 8 on its port o_1 at time point 1, even if deployed to the architecture. For *dynamic* architectures, on the other hand, the traditional interpretation of temporal specifications of the behavior of components is not valid anymore. For example, it is not clear whether the trace depicted in Fig. 1 actually fulfills the original specification of component c_3, since c_3 is not active at time point 1 ($n = 1$).

So, how can we reason about the behavior of components deployed to dynamic architectures? To answer this question, in the following we provide a calculus for dynamic architectures. It formalizes reasoning about the behavior of a component when it can be activated and deactivated. In the spirit of natural deduction, we provide introduction and elimination rules for each temporal operator. Finally, we show *soundness* and relative *completeness* of the calculus. As a practical implication, our calculus can be used to support the verification of properties for dynamic architectures. This is demonstrated by means of the blackboard pattern for dynamic architectures. To this end, we apply the calculus to verify a characteristic property of the pattern.

The remainder of the paper is structured as follows: First, we introduce our model for dynamic architectures in Sect. 2. In Sect. 3, we then provide the notion of behavior assertions and behavior trace assertions as means to specify the behavior of components. In Sect. 4, we introduce our calculus, which allows us to reason about component behavior in a dynamic context. Sect. 5 then demonstrates the practical usability of the calculus by applying it to verify a property of dynamic blackboard architectures. Finally, we conclude our discussion with a review of related work in Sect. 6 and a brief summary of the major contributions of this paper in Sect. 7.

2 A Model of Dynamic Architectures

In [15], we introduce a model for dynamic architectures based on the notion of configuration traces. Our model is based on Broy's Focus theory [3] and an adaptation of its dynamic extension [4]. In this section, we briefly summarize the main concepts of the model and extend it with the notion of behavior traces to model the behavior of single components.

2.1 Foundations: Ports, Valuations, and Components

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

Port Valuations. Ports can be valuated by messages. Roughly speaking, a valuation for a set of ports is an assignment of messages to each port.

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

$$\overline{P} \overset{def}{=} (P \to \wp(\mathsf{M})).$$

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 $[p_1, p_2, \ldots \mapsto m_1, m_2, \ldots]$.

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.

Components. In our model, the basic unit of computation is a component. It consists of an identifier and a set of input and output ports. Thus, we assume the existence of set C_{id} containing all component identifiers.

Definition 2 (Component). *A* component *is a triple (id, I, O) consisting of:*

- *a component identifier $id \in \mathsf{C}_{id}$ and*
- *two* disjoint *sets of input and output ports $I, O \subseteq \mathsf{P}$.*

The set of all components is denoted by $\mathcal{C}$. For a set of components $C \subseteq \mathcal{C}$, we denote by:

- $\mathsf{in}(C) \overset{def}{=} \bigcup_{(id,I,O)\in C}(\{id\} \times I)$ *the set of* component input ports*,*
- $\mathsf{out}(C) \overset{def}{=} \bigcup_{(id,I,O)\in C}(\{id\} \times O)$ *the set of* component output ports*,*
- $\mathsf{port}(C) \overset{def}{=} \mathsf{in}(C) \cup \mathsf{out}(C)$ *the set of all* component ports*, and*
- $\mathsf{id}(C) \overset{def}{=} \bigcup_{(id,I,O)\in C}\{id\}$ *the set of all* component identifiers.

A set of components $C \subseteq \mathcal{C}$ is called healthy *iff a component is uniquely determined by its name:*

$$\forall (id, I, O), (id', I', O') \in C\colon id = id' \implies I = I' \wedge O = O'. \qquad (1)$$

Similar to Definition 1, we define the set of all possible component port valuations (CPVs) *for a set of* component $P \subseteq \mathsf{C}_{id} \times \mathsf{P}$.

2.2 Modeling Component Behavior

A component's behavior is modeled by a set of execution traces over the component's interface. In the following, we denote with $(E)^+$ the set of all finite sequences over elements of a given set E, by $(E)^\infty$ the set of all infinite sequences over E, and by $(E)^*$ the set of all finite and infinite sequences over E.

Definition 3 (Behavior Trace). *A behavior trace for a component* (id, I, O) *is an infinite sequence* $(\overline{I \times O})^\infty$. *The set of all BTs for component c is denoted by* $\mathcal{B}(c)$.

Note that a component's behavior is actually modeled as a *set* of behavior traces, rather than just a single trace. This is to handle non-determinism for inputs to, as well as outputs from components.

Example 1 (Behavior Trace). In the following, we provide a possible BT for a component c_3 with two input ports i_0 and i_1, and two output ports o_0 and o_1: $[i_0, i_1, o_0, o_1 \mapsto X, 5, 9, \{8, 4\}]$, $[i_0, i_1, o_0, o_1 \mapsto \{T, B\}, \{2, 4\}, 7, \{3, 9\}], \cdots$.

2.3 Modeling Dynamic Architectures

Dynamic architectures are modeled as sets of *configuration traces* which are sequences over *architecture configurations.*

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

Definition 4 (Architecture Configuration). *An architecture configuration (AC) over a* healthy *set of components* $C \subseteq \mathcal{C}$ *is a triple* (C', N, μ), *consisting of:*

- *a set of active components* $C' \subseteq C$,
- *a connection* $N \colon \mathsf{in}(C') \to \wp(\mathsf{out}(C'))$, *and*
- *a CPV* $\mu \in \overline{\mathsf{port}(C')}$.

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

$$\forall p_i \in \mathsf{in}(C') \colon N(p_i) \neq \emptyset \implies \mu(p_i) = \bigcup_{p_o \in N(p_i)} \mu(p_o). \tag{2}$$

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

Note that connection N is modeled as a set-valued function from component input ports to component output ports, meaning that: (i) input/output ports can be connected to several output/input ports, respectively, and (ii) not every input/output port needs to be connected to an output/input port (in which case the connection returns the empty set).

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

Definition 5 (Configuration Trace). *A configuration trace (CT) over a healthy set of components $C \subseteq \mathcal{C}$ is an infinite sequence $(\mathcal{K}(C))^\infty$. The set of all CTs over C is denoted by $\mathcal{R}(C)$.*

Example 2 (Configuration Trace). Fig. 1 shows the first three ACs of a possible CT. The first AC, $t(0) = (C', N, \mu)$, e.g., consists of:

- components $C' = \{C_1, C_2, C_3\}$, with $C_1 = (c_1, \{i_0\}, \{o_0, o_1, o_2\})$, $C_2 = (c_2, \{i_0, i_1, i_2\}, \{o_0\})$, and $C_3 = (c_3, \{i_0, i_1\}, \{o_0, o_1\})$;
- connection N, with $N((c_2, i_1)) = \{(c_1, o_1)\}$, $N((c_3, i_1)) = \{(c_1, o_2)\}$, and $N((c_2, i_2)) = \{(c_3, o_1)\}$; and
- valuation $\mu = [(c_1, i_0), (c_1, o_0), (c_2, i_2), \cdots \mapsto 5, 9, \{8, 4\}, \cdots]$.

Note that a dynamic architecture is modeled as a *set* of CTs rather than just one single trace. Again, this allows for non-determinism in inputs to an architecture as well as its reaction. Moreover, note that our notion of architecture is dynamic in the following sense: (i) *components* may appear and disappear over time and (ii) *connections* may change over time.

In the following, we introduce an operator to denote the number of activations of a component in a (possible finite) configuration trace. Thereby, we denote by $[c]^i = c_i$ the i-th component (where $i \geq 1$ and $i \leq n$) of a tuple $c = (c_1, \ldots, c_n)$.

Definition 6 (Number of Activations). *With $\langle c \overset{n}{\#} t\rangle$, we denote the number of* activations *of component c in a (possibly finite) configuration trace t up to (excluding) point in time n:*

$$
\begin{aligned}
\langle c \overset{0}{\#} t\rangle &\overset{def}{=} 0,\\
c \in [t(n)]^1 \implies \langle c \overset{n+1}{\#} t\rangle &\overset{def}{=} \langle c \overset{n}{\#} t\rangle + 1,\\
c \notin [t(n)]^1 \implies \langle c \overset{n+1}{\#} t\rangle &\overset{def}{=} \langle c \overset{n}{\#} t\rangle.
\end{aligned}
$$

Moreover, we introduce an operator to return the last activation of a component in a configuration trace.

Definition 7 (Last Activation). *With $last(t, c)$, we denote the* greatest $i \in \mathbb{N}$, *such that $c \in [t(i)]^1$.*

Note that $last(t, c)$ is well-defined iff $\exists i \in \mathbb{N} \colon c \in [t(i)]^1$ and $\exists n \in \mathbb{N} \colon \forall n' \geq n \colon c \notin [t(n')]^1$.

Finally, we introduce an operator which for a given point in time returns the least earlier point in time where a certain component was not yet active.

Definition 8 (Least Not Active). *With $\langle c \overset{n}{\vee} t\rangle$, we denote the* least $n' \in \mathbb{N}$, *such that $n' = n \vee \left(n' < n \wedge \forall n' \leq \underset{.}{k} \leq n \colon c \notin [t(n')]^1\right)$.*

Note that $\langle c \overset{n}{\vee} t\rangle$ is always well-defined and for the case in which $c \in [t(n)]^1$, it returns n itself.

2.4 From Configuration Traces to Behavior Traces

In the following, we introduce the notion of projection to extract the behavior of a certain component out of a given CT.

Definition 9 (Projection). *Given a (finite or infinite) CT$t \in (\mathcal{K}(C))^*$ over a healthy set of components $C \subseteq \mathcal{C}$. The projection to component $c = (id, I, O) \in C$ is denoted by $\Pi_c(t) \in (\mathcal{B}(c))^*$ and defined as the* greatest *relation satisfying the following equations:*

$$\begin{aligned} \Pi_c(t\,|_0) &\overset{def}{=} \langle\rangle, \\ c \in [t(n)]^1 \implies \Pi_c(t\,|_{n+1}) &\overset{def}{=} \Pi_c(t\,|_n) \mathbin{\widehat{}} \big(\lambda p \in I \cup O\colon\ [t(n)]^3\,(id, p)\big), \\ c \notin [t(n)]^1 \implies \Pi_c(t\,|_{n+1}) &\overset{def}{=} \Pi_c(t\,|_n), \end{aligned}$$

where $s \mathbin{\widehat{}} e$ denotes the sequence resulting from appending element e to sequence s.

Example 3 (Projection). Applying projection of component c_3 to the CT given by Example 2 results in a BT starting as described by Example 1.

Note that for systems in which a component is activated only finitely many times, the projection to this component results in only a finite behavior trace.

3 Specifying Component Behavior

In the following, we introduce the notion of *behavior trace assertions*, a language to specify component behavior over a given interface specification. We provide its syntax as well as a formal semantics thereof in terms of *behavior traces*. Finally, we introduce a satisfaction relation for configuration traces which serves as a foundation for the calculus presented in the next section.

3.1 Behavior Trace Assertions

Component behavior can be specified by means of behavior trace assertions, i.e., temporal logic [13] formulæ over behavior assertions. Behavior assertions, on the other hand, are used to specify a component's state at a certain point in time. They are specified over a given interface specification.

Interface Specifications. Interfaces declare a set of port identifiers and associate a sort with each port. Thus, in the following, we postulate the existence of the set of all port identifiers P_{id}. Moreover, interfaces are specified over a given signature $\Sigma = (S, F, B)$ consisting of a set of sorts S, function symbols F, and predicate symbols B.

Definition 10 (Interface Specification). *An interface specification (IS) over a signature $\Sigma = (S, F, B)$ is a triple (P_{in}, P_{out}, t^p), consisting of:*

- *two* disjoint *sets of input and output port identifiers $P_{in}, P_{out} \subseteq \mathsf{P}_{id}$,*
- *a mapping $t^p\colon P_{in} \cup P_{out} \rightarrow S$ assigning a sort to each port identifier.*

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

Behavior Assertions. Behavior assertions specify a component's state (i.e.: valuations of its ports with messages) at a certain point in time. In the following, we do not go into the details of how to specify such assertions, rather, we assume the existence of a set containing all type-compatible behavior assertions over a given interface specification.

Definition 11 (Behavior Assertions). *Given IS $S_i = (P_{in}, P_{out}, t^p)$ over signature $\Sigma = (S, F, B)$ and family of variables $V = (V_s)_{s \in S}$ with variables V_s for each sort $s \in S$. With $\varphi_\Sigma^V(S_i)$, we denote the set of all* type-compatible *(with regard to t^p) behavior assertions (BAs) for S_i, Σ, and V.*

Algebras and Variable Assignments. A BA is always interpreted over a given algebra for the signature used in the corresponding IS. Thus, in the following, we denote by $\mathcal{A}(\Sigma)$ the set of all algebras $(S', F', B', \alpha, \beta, \gamma)$ for signature $\Sigma = (S, F, B)$, consisting of sets S', functions F', predicates B', and corresponding interpretations $\alpha \colon S \to S'$, $\beta \colon F \to F'$, and $\gamma \colon B \to B'$. Moreover, with $\mathcal{I}_A^V$, we denote the set of all *variable assignments (VAs)* $\iota = (\iota_s)_{s \in S}$ (with $\iota_s \colon V_s \to \alpha(s)$ for each $s \in S$) for a family of variables $V = (V_s)_{s \in S}$ in an algebra A.

Semantics of Behavior Assertions. The semantics of behavior assertions is described in terms of component port valuations satisfying a certain behavior assertion. In the following, we denote with $A \leftrightarrow B$ a bijective function from set A to set B.

Definition 12 (Behavior Assertions: Semantics). *Given interface specification $S_i = (P_{in}, P_{out}, t^p) \in \mathcal{S}_I(\Sigma)$, a healthy set of components $C \subseteq \mathcal{C}$, component $c = (id, I, O) \in C$, algebra $A \in \mathcal{A}(\Sigma)$, and VA$\iota = (\iota_s)_{s \in S} \in \mathcal{I}_A^V$. We denote with $\mu \ {}_b{\models}_{(A,\iota)}^{(\delta^i,\delta^o)} \gamma$ that $\mu \in \overline{I \cup O}$ satisfies BA $\gamma \in \varphi_\Sigma^V(S_i)$ for port interpretations (PIs) $\delta^i \colon I \leftrightarrow P_{in}$ and $\delta^o \colon O \leftrightarrow P_{out}$.*

Behavior Trace Assertions. Behavior trace assertions are a means to specify a component's behavior in terms of temporal specifications over behavior assertions.

Definition 13 (Behavior Trace Assertions). *For a family of variables $V = (V_s)_{s \in S}$,* rigid *(fixed for the whole execution) variables $V' = (V'_s)_{s \in S}$, the set of all behavior trace assertions (BTAs) for IS$S_i \in \mathcal{S}_I(\Sigma)$ is given by $\Gamma_\Sigma^{(V,V')}(S_i)$ and defined inductively by the equations provided in Fig. 2.*

$$
\begin{aligned}
\phi \in \varphi_\Sigma^{V \cup V'}(S_i) &\implies \phi \in \Gamma_\Sigma^{(V,V')}(S_i) \ , \\
\text{“}\gamma\text{”} \in \Gamma_\Sigma^{(V,V')}(S_i) &\implies \text{“}\bigcirc \gamma\text{”},\ \text{“}\Diamond\gamma\text{”},\ \text{“}\Box\gamma\text{”} \in \Gamma_\Sigma^{(V,V')}(S_i) \ , \\
\text{“}\gamma\text{”}, \text{“}\gamma'\text{”} \in \Gamma_\Sigma^{(V,V')}(S_i) &\implies \text{“}(\gamma \ \mathcal{U} \ \gamma')\text{”} \in \Gamma_\Sigma^{(V,V')}(S_i) \ .
\end{aligned}
$$

Fig. 2. Inductive definition of behavior trace assertions.

3.2 Semantics: Behavior Traces

In the following, we define what it means for a behavior trace to satisfy a corresponding behavior trace assertion.

Definition 14 (Semantics BTs). *Given algebra A and corresponding VAs $\iota' = (\iota'_s)_{s \in S} \in \mathcal{I}_A^{V'}$ for variables V'. With $(t,n)\ {}^{t}_{b}{\models}^{(\delta^i,\delta^o)}_{(A,\iota')} \gamma$, defined recursively by the equations listed in Fig. 3, we denote that BT $t \in \mathcal{B}(c)$ satisfies BA $\gamma \in \Gamma_\Sigma^{(V,V')}(S_i)$ at time $n \in \mathbb{N}$. A BT $t \in \mathcal{B}(c)$ satisfies BA $\gamma \in \Gamma_\Sigma^{(V,V')}(S_i)$, denoted $t\ {}^{t}_{b}{\models}^{(\delta^i,\delta^o)}_{(A,\iota')} \gamma$ iff $(t,0)\ {}^{t}_{b}{\models}^{(\delta^i,\delta^o)}_{(A,\iota')} \gamma$.*

$$\begin{aligned}
(t,n)\ {}^{t}_{b}{\models}^{(\delta^i,\delta^o)}_{(A,\iota')} \text{“}\phi\text{”} &\iff \forall \iota \in \mathcal{I}_A^V \colon t(n)\ {}_{b}{\models}^{(\delta^i,\delta^o)}_{(A,\iota\cup\iota')} \text{“}\phi\text{”}\ \text{[for } \phi \in \varphi_\Sigma^V(S_i)\text{]}\ , \\
(t,n)\ {}^{t}_{b}{\models}^{(\delta^i,\delta^o)}_{(A,\iota')} \text{“}\bigcirc \gamma\text{”} &\iff (t,n+1)\ {}^{t}_{b}{\models}^{(\delta^i,\delta^o)}_{(A,\iota')} \text{“}\gamma\text{”}\ , \\
(t,n)\ {}^{t}_{b}{\models}^{(\delta^i,\delta^o)}_{(A,\iota')} \text{“}\Diamond\gamma\text{”} &\iff \exists n' \geq n \colon (t,n')\ {}^{t}_{b}{\models}^{(\delta^i,\delta^o)}_{(A,\iota')} \text{“}\gamma\text{”}\ , \\
(t,n)\ {}^{t}_{b}{\models}^{(\delta^i,\delta^o)}_{(A,\iota')} \text{“}\Box\gamma\text{”} &\iff \forall n' \geq n \colon (t,n')\ {}^{t}_{b}{\models}^{(\delta^i,\delta^o)}_{(A,\iota')} \text{“}\gamma\text{”}\ , \\
(t,n)\ {}^{t}_{b}{\models}^{(\delta^i,\delta^o)}_{(A,\iota')} \text{“}(\gamma'\ \mathcal{U}\ \gamma)\text{”} &\iff \exists n' \geq n \colon (t,n')\ {}^{t}_{b}{\models}^{(\delta^i,\delta^o)}_{(A,\iota')} \text{“}\gamma\text{”}\ \wedge \\
&\qquad \forall n \leq m < n' \colon (t,m)\ {}^{t}_{b}{\models}^{(\delta^i,\delta^o)}_{(A,\iota')} \text{“}\gamma'\text{”}\ .
\end{aligned}$$

Fig. 3. Recursive definition of satisfaction relation for behavior traces.

3.3 Semantics: Configuration Traces

In the following, we define what it means for a configuration trace to satisfy a behavior assertion.

Definition 15 (Semantics CTs). *Given algebra A, corresponding VAs $\iota' = (\iota'_s)_{s \in S} \in \mathcal{I}_A^{V'}$ for variables V', and behavior trace $t' \in \mathcal{B}(c)$. With*

$$\begin{aligned}
&(t,t',n)\ {}^{t}_{k}{\models}^{(c,\delta^i,\delta^o)}_{(A,\iota')} \gamma \overset{def}{\iff} \\
&\quad \Big(\exists i \geq n \colon c \in [t(i)]^1 \wedge \big(\Pi_c(t) \circ t', \langle c \overset{n}{\#} t\rangle\big)\ {}^{t}_{b}{\models}^{(\delta^i,\delta^o)}_{(A,\iota')} \gamma\Big) \quad \vee && (3) \\
&\quad \Big(\exists i \colon c \in [t(i)]^1 \wedge \nexists i \geq n \colon c \in [t(i)]^1 \wedge \\
&\qquad\quad \big(\Pi_c(t) \circ t', \#(\Pi_c(t)) - 1 + (n - \mathit{last}(t,c))\big)\ {}^{t}_{b}{\models}^{(\delta^i,\delta^o)}_{(A,\iota')} \gamma\Big) \quad \vee && (4) \\
&\quad \Big(\nexists i \colon c \in [t(i)]^1 \wedge \big(t', n\big)\ {}^{t}_{b}{\models}^{(\delta^i,\delta^o)}_{(A,\iota')} \gamma\Big), && (5)
\end{aligned}$$

we denote that CT $t \in \mathcal{R}(C)$ satisfies BA $\gamma \in \Gamma_\Sigma^{(V,V')}(S_i)$ at time $n \in \mathbb{N}$ for a given continuation t'. Again, a CT $t \in \mathcal{B}(c)$ satisfies BA $\gamma \in \Gamma_\Sigma^{(V,V')}(S_i)$, denoted $t\ {}^{t}_{k}{\models}^{(c,\delta^i,\delta^o)}_{(A,\iota')} \gamma$ iff $(t,0)\ {}^{t}_{k}{\models}^{(c,\delta^i,\delta^o)}_{(A,\iota')} \gamma$.

To satisfy a given behavior assertion γ for a component c at a certain point in time n under a given continuation t', a configuration trace t is required to fulfill one of the following conditions:

- By Eq. (3): Component c is again activated (after time point n) and the projection to c for t fulfills γ at the point in time given by the current number of activations of c.
- By Eq. (4): Component c is activated at least once but not again in the future and the continuation fulfills γ at the point in time resulting from the difference of the current point in time and the last activation of c.
- By Eq. (5): Component c is never activated and the continuation fulfills γ at point in time n.

For the sake of readability, from now on, we omit symbols for algebras and port/variable interpretations for satisfaction relations. An algebra and corresponding interpretations are, however, assumed to be fixed for each property.

The following property ensures correctness of Definition 15:

Proposition 1 (Soundness of Definition 15). *A CT* $t \in \mathcal{R}(c)$ *satisfies BA* $\gamma \in \Gamma_{\Sigma}^{(V,V')}(S_i)$ *for a given continuation* $t' \in \mathcal{B}(c)$ *iff the corresponding projection satisfies* γ*:*

$$(t,t') \; {}^{t}_{k}{\models}_{(c)} \gamma \iff \Pi_c(t) \circ t' \; {}^{t}_{b}{\models} \gamma,$$

where $s \circ s'$ *denotes the sequence resulting from concatenating sequences* s *and* s'*.*

Remember that for architectures in which a component is activated only finitely many times, the projection to this component results in only a finite behavior trace. This is why we actually check for a valid continuation $t' \in \mathcal{B}(c)$.

4 A Calculus for Dynamic Architectures

Until now, ${}^{t}_{k}{\models}$ is only implicitly defined in terms of ${}^{t}_{b}{\models}$. While this mirrors our intuition about ${}^{t}_{k}{\models}$, it is not very useful to reason about it. Thus, in the following section, we provide an explicit characterization of ${}^{t}_{k}{\models}$ in terms of a calculus for dynamic architectures. Then, we show *soundness* and relative *completeness* of the calculus with regard to Definition 15. Using a natural deduction style, we provide introduction and elimination rules for each temporal operator.

4.1 Introduction Rules

We provide 8 rules which can be used to introduce temporal operators in a dynamic context.

Behavior Assertions. The first rules characterize introduction for *basic* behavior assertions. Therefore, we distinguish between three cases: First, the following case in which a component is guaranteed to be eventually activated in the future:

AssI$_\text{a}$

$$\frac{\begin{array}{c}\left[n \le i \quad c \in [t(i)]^1 \quad \nexists n \le k < i\colon c \in [t(k)]^1\right]\\ \vdots \\ \lambda p \in I \cup O\colon\ [t(i)]^3\,(c,p)\ {}_{b}{\models} \text{“}\phi\text{”}\end{array}}{(t,t',n)\ {}^{\,t}_{k}{\models}_{(c)} \text{“}\phi\text{”}}\ \exists i \ge n\colon c \in [t(i)]^1$$

For this case, in order to show that a BA ϕ holds at time point n, we have to show that ϕ holds at the very next point in time at which component c is active.

For the case in which a component was sometimes active, but is not activated again in the future, we get the following rule:

AssI$_\text{n1}$

$$\frac{t'\big(n - \mathit{last}(c,t)\big)\ {}_{b}{\models} \text{“}\phi\text{”}}{(t,t',n)\ {}^{\,t}_{k}{\models}_{(c)} \text{“}\phi\text{”}}\ \exists i\colon c \in [t(i)]^1 \wedge \nexists i \ge n\colon c \in [t(i)]^1$$

In order to show that BA ϕ holds at a certain point in time n, we have to show that ϕ holds for the continuation t'. Note that the corresponding time point is calculated as the difference from n to the last point in time at which component c was active in t.

Finally, we have another rule for the case in which component is never activated:

AssI$_\text{n2}$

$$\frac{t'(n)\ {}_{b}{\models} \text{“}\phi\text{”}}{(t,t',n)\ {}^{\,t}_{k}{\models}_{(c)} \text{“}\phi\text{”}}\ \nexists i\colon c \in [t(i)]^1$$

For such cases, BA ϕ holds at a certain point in time n when ϕ holds for t' at time point n.

Next. The next rule characterizes introduction for the *next* operator. For this operator as well, we distinguish two cases: The first case is again the one in which a component is guaranteed to be eventually activated in the future:

NxtI$_\text{a}$

$$\frac{\begin{array}{c}\left[n \le i \quad c \in [t(i)]^1 \quad \nexists n \le k < i\colon c \in [t(k)]^1\right]\\ \vdots \\ (t,t',i+1)\ {}^{\,t}_{k}{\models}_{(c)} \text{“}\gamma\text{”}\end{array}}{(t,t',n)\ {}^{\,t}_{k}{\models}_{(c)} \text{“}\bigcirc\gamma\text{”}}\ \exists i \ge n\colon c \in [t(i)]^1$$

For this case, in order to show that a BTA $\bigcirc\gamma$ holds at a certain point in time n, we have to show that it holds *after* the very next activation of c in t.

For the case in which a component is not activated again in the future, we get the following rule for the *next* operator:

NxtI$_\mathrm{n}$

$$\frac{(t,t',n+1)\ {}^{t}_{k}{\models}_{(c)}\ \text{“}\gamma\text{”}}{(t,t',n)\ {}^{t}_{k}{\models}_{(c)}\ \text{“}\bigcirc\gamma\text{”}}\ \nexists i\geq n\colon c\in[t(i)]^{1}$$

In this case, the dynamic interpretation of the operator resembles its traditional one. Thus, it suffices to show that BTA γ holds for the *next* point in time $n+1$, in order to conclude that $\bigcirc\gamma$ holds at n.

Eventually. Introduction for the *eventually* operator can be described with a single rule:

EvtI

$$\frac{\langle c \overset{n}{\vee} t\rangle \leq n' \qquad (t,t',n')\ {}^{t}_{k}{\models}_{(c)}\ \text{“}\gamma\text{”}}{(t,t',n)\ {}^{t}_{k}{\models}_{(c)}\ \text{“}\diamondsuit\gamma\text{”}}$$

It states that in order to show that $\diamondsuit\gamma$ holds for a component c at some point in time n, we only have to show that γ holds at *some* time point later than the last activation (before n) of c.

Globally. Similarly, we provide a single introduction rule for the *globally* operator:

GlobI

$$\frac{\begin{array}{c}[n\leq n']\\ \vdots\\ (t,t',n')\ {}^{t}_{k}{\models}_{(c)}\ \text{“}\gamma\text{”}\end{array}}{(t,t',n)\ {}^{t}_{k}{\models}_{(c)}\ \text{“}\Box\gamma\text{”}}$$

It allows us to conclude $\Box\gamma$ for time point n whenever we can show that γ holds for an *arbitrary* $n'\geq n$.

Until. Finally, we provide a single rule for introducing the *until* operator:

UntilI

$$\frac{\langle c\overset{n}{\vee}t\rangle\leq n' \quad (t,t',n')\ {}^{t}_{k}{\models}_{(c)}\ \text{“}\gamma\text{”} \quad \begin{array}{c}\begin{bmatrix} n\leq n'' & n''\leq i''\\ c\in[t(i'')]^{1} & i''<n'\end{bmatrix}\\ \vdots\\ (t,t',n'')\ {}^{t}_{k}{\models}^{(c,\delta^{i},\delta^{o})}_{(A,\iota)}\ \text{“}\gamma'\text{”}\end{array} \begin{array}{c}\begin{bmatrix} n\leq n'' & n''<n'\\ \nexists i''\geq n''\colon & c\in[t(i'')]^{1}\end{bmatrix}\\ \vdots\\ (t,t',n'')\ {}^{t}_{k}{\models}^{(c,\delta^{i},\delta^{o})}_{(A,\iota)}\ \text{“}\gamma'\text{”}\end{array}}{(t,t',n)\ {}^{t}_{k}{\models}_{(c)}\ \text{“}\gamma'\ \mathcal{U}\ \gamma\text{”}}$$

In order to show that $\gamma' \ \mathcal{U} \ \gamma$ holds for a component c at some point in time n, the rule requires to show that γ holds at some point n' later than the last activation (before n) of c and that for every time point *up to* the last activation of component c *before* n' (or the last time point $n'' < n'$ for the case component c is not activated anymore), γ' holds.

4.2 Elimination Rules

In contrast to introduction, we provide 10 rules for the elimination of the different temporal operators.

Behavior Assertions. Again, we first provide rules characterizing elimination for *basic* behavior assertions. Similar to introduction, we distinguish between three cases: The first case describes elimination for situations in which a component is guaranteed to be activated sometimes in the future:

AssE$_\text{a}$

$$\frac{(t,t',n)\ {}^{t}_{k}\!\models_{(c)} \text{“}\phi\text{”} \quad n \leq i \quad c \in [t(i)]^1 \quad \nexists n \leq k < i \colon c \in [t(k)]^1}{\lambda p \in I \cup O \colon\ [t(i)]^3 (c,p)\ {}_{b}\!\models \text{“}\phi\text{”}}\ \exists i \geq n \colon c \in [t(i)]^1$$

The rule for such cases allows us to eliminate a basic BA ϕ and conclude that ϕ holds at the very *next* point in time where component c is active.

The next rule deals with the case in which a component was sometimes active, but is not activated again in the future.

AssE$_\text{n1}$

$$\frac{(t,t',n)\ {}^{t}_{k}\!\models_{(c)} \text{“}\phi\text{”}}{t'\big(n - \mathit{last}(c,t)\big)\ {}_{b}\!\models \text{“}\phi\text{”}}\ \exists i \colon c \in [t(i)]^1 \wedge \nexists i \geq n \colon c \in [t(i)]^1$$

The rule for this case allows us to conclude that a BA ϕ holds at a certain point in time for continuation t'. Again, the corresponding time point is calculated as the difference of n and the last time component c was activated.

Finally, we have another rule for the case in which component is never activated:

AssE$_\text{n2}$

$$\frac{(t,t',n)\ {}^{t}_{k}\!\models_{(c)} \text{“}\phi\text{”}}{t'(n)\ {}_{b}\!\models \text{“}\phi\text{”}}\ \nexists i \colon c \in [t(i)]^1$$

For such cases, we may eliminate ϕ and conclude that ϕ holds at n for continuation t'.

Next. Similar to introduction, we provide two rules to eliminate a *next* operator: The first rule deals again with the case in which a component is guaranteed to be activated sometimes in the future:

NxtE$_{\mathrm{a}}$

$$\frac{(t,t',n)\;{}^{t}_{k}\!\models_{(c)}\text{“}\bigcirc\gamma\text{”}\quad n\leq i\quad c\in[t(i)]^{1}\quad \nexists n\leq k<i\colon c\in[t(k)]^{1}}{(t,t',i+1)\;{}^{t}_{k}\!\models_{(c)}\text{“}\gamma\text{”}}\;\exists i\geq n\colon c\in[t(i)]^{1}$$

Similar to the corresponding introduction rule, this rule allows us to conclude BTA γ for a certain point in time $i+1$, whenever $\bigcirc\gamma$ holds at an earlier point in time n and i is the very next activation of component c.

If a component is not activated again, we get the following rule for eliminating a *next* operator:

NxtE$_{\mathrm{n}}$

$$\frac{(t,t',n)\;{}^{t}_{k}\!\models_{(c)}\text{“}\bigcirc\gamma\text{”}}{(t,t',n+1)\;{}^{t}_{k}\!\models_{(c)}\text{“}\gamma\text{”}}\;\nexists i\geq n\colon c\in[t(i)]^{1}$$

Again, the rule resembles the traditional interpretation of *next*, which allows us to conclude that BTA γ holds for a certain point in time $n+1$, whenever $\bigcirc\gamma$ holds at n.

Eventually. We provide two rules to eliminate an *eventually* operator:

EvtE$_{\mathrm{a}}$

$$\frac{(t,t',n)\;{}^{t}_{k}\!\models_{(c)}\text{“}\Diamond\gamma\text{”}}{\exists n'\geq\langle c\overset{n}{\vee}t\rangle\colon(t,t',n')\;{}^{t}_{k}\!\models_{(c)}\text{“}\gamma\text{”}}\;\exists i\geq n\colon c\in[t(i)]^{1}$$

When eliminating a $\Diamond\gamma$ for a component c at time point n, the rule allows us to conclude that BTA γ holds sometimes after the last activation (before n) of component c.

A similar rule applies for the case in which c is not activated again ($\exists n'\geq n\colon(t,t',n')\;{}^{t}_{k}\!\models_{(c)}\text{“}\gamma''$). For this case (denoted EvtE$_{\mathrm{n}}$), however, we can conclude that the corresponding point in time n' is actually greater than n instead of $\langle c\overset{n}{\vee}t\rangle$.

Globally. Similar to introduction, we have a single rule for the elimination of a globally operator:

GlobE

$$\frac{(t,t',n)\;{}^{t}_{k}\!\models_{(c)}\text{“}\Box\gamma\text{”}\quad n'\geq\langle c\overset{n}{\vee}t\rangle}{(t,t',n')\;{}^{t}_{k}\!\models_{(c)}\text{“}\gamma\text{”}}$$

The rule allows us to eliminate $\Box\gamma$ for component c at time point n and conclude that γ holds at an arbitrary point later than the last activation of c before n.

Until. Finally, we provide two rules to eliminate *until* operators:

UntilE$_\text{a}$

$$\frac{(t,t',n)\ {}^{\,t}_{k}{\models}_{(c)}\ \text{“}\gamma'\ \mathcal{U}\ \gamma\text{”}}{\begin{array}{l}\exists n' \geq \langle c \overset{n}{\#} t\rangle\colon (t,t',n')\ {}^{\,t}_{k}{\models}_{(c)}\ \text{“}\gamma\text{”}\ \wedge\\ \Big(\forall n'' \geq \langle c \overset{n}{\vee} t\rangle\colon \big(\exists n'' \leq i' < n'\colon c \in [t(i')]^1\,\big)\\ \quad \vee\ (\nexists i \geq n''\colon c \in [t(i)]^1 \wedge n'' < n')\\ \qquad \Longrightarrow\ (t,t',n'')\ {}^{\,t}_{k}{\models}_{(c)}\ \text{“}\gamma'\text{”}\Big)\end{array}}\ \exists i \geq n\colon c \in [t(i)]^1$$

Assuming that $\gamma'\ \mathcal{U}\ \gamma$ holds at some time point n, the rule allows us to conclude that there exists a time point in the future n', such that BTA γ holds and that up to the last activation of component c earlier to n' (or the last time point $n'' < n'$ for the case component c is not activated anymore), BTA γ' holds.

Again, a similar rule applies for the case in which c is not activated again ($\exists n' \geq n\colon (t,t',n')\ {}^{\,t}_{k}{\models}_{(c)}\ \text{“}\gamma''$). For this case (denoted UntilE$_\text{n}$), however, we can conclude that the corresponding point in time n' is actually greater than n instead of $\langle c \overset{n}{\vee} t\rangle$.

4.3 Soundness and Completeness

In the following, we show *soundness* and relative *completeness* of the calculus. Thereby, we denote with $\vdash_{DA}\big((t,n)\ {}^{\,t}_{k}{\models}_{(c)}\gamma\big)$ that it is possible to derive $(t,n)\ {}^{\,t}_{k}{\models}_{(c)}\gamma$ with the rules introduced in Sect. 4. With $\models_{DA}\big((t,n)\ {}^{\,t}_{k}{\models}_{(c)}\gamma\big)$, on the other hand, we denote that configuration trace t indeed satisfies BTA γ at time point n.

Theorem 1 (Soundness). *The calculus presented in Sects. 4.1 and 4.2 is sound:*

Proof (Sketch). For each rule, we assume its premises and prove its conclusions from Definitions 14 and 15.

Theorem 2 (Completeness). *The calculus presented in Sects. 4.1 and 4.2 is complete (relative to the completeness of ${}_{b}{\models}$):*

$$\vdash_{DA}\Big((t,n)\ {}^{\,t}_{k}{\models}_{(c)}\ \text{“}\gamma\text{”}\Big) \qquad \Longleftarrow \qquad \models_{DA}\Big((t,n)\ {}^{\,t}_{k}{\models}_{(c)}\ \text{“}\gamma\text{”}\Big).$$

Proof (Sketch). The validity of each BTA can be derived by applying the corresponding introduction rules.

5 Verifying Properties of Dynamic Architectures

In the following, we demonstrate the practical usability of the calculus presented in Sect. 4. Therefore, we specify a dynamic version of the blackboard architecture pattern and apply our calculus to verify a simple property of such architectures.

5.1 Dynamic Blackboard Architectures: Specification

In the following, we introduce a simplified version of the blackboard pattern as described, for example, by Shaw and Garlan [18], Buschmann et al. [5], and Taylor et al. [19]. Therefore, we first specify the involved datatypes, the components interfaces, and constraints regarding the activation/deactivation of components as well as connections between their ports. Then we provide a specification of component behavior in terms of BTAs.

Datatypes. Blackboard architectures work with *problems* and *solutions* for them. Figure 4a provides the corresponding datatype specification (DTS) in terms of an algebraic specification [21]. We denote by `PROB` the set of all problems and by `SOL` the set of all solutions. To relate a problem with a corresponding solution, we assume the existence of a function s: `PROB` → `SOL` which assigns the *correct* solution to each problem.

Interfaces. In our example, a blackboard architecture consists of a blackboard (BB) component and a knowledgesource (KS) component. The configuration diagram (CD) [14] in Fig. 4c shows the specification of the corresponding interfaces. In our simple example, the BB component merely forwards messages to and from the KS component. Thus, it has an input port i_p which receives a problem and an output port o_s which returns the corresponding solution. Moreover, it has an output port p_p to forward a problem to a KS and a corresponding input port p_s to receive its solution. A KS, in our example, gets a problem on its input port p_p and provides a corresponding solution on its output port p_s.

DTSpec ProbSol	
sort `PROB`, `SOL`	
s :	`PROB` → `SOL`

(a) Datatype Specification.

PSpec BPort	**uses** ProbSol
i_p, p_p :	`PROB`
o_s, p_s :	`SOL`

(b) Port Specification.

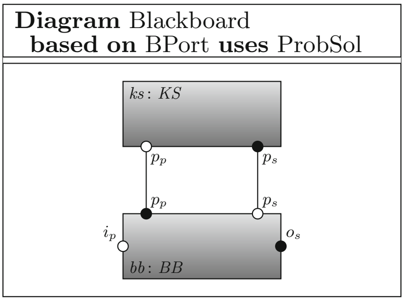

Diagram Blackboard
based on BPort **uses** ProbSol

(c) Configuration Diagram.

Fig. 4. Specification of the blackboard pattern.

Spec Blackboard_Activation **uses** *Blackboard*

$$\Box\Big(bb.p_p \neq \emptyset \implies \|ks\|\Big) \tag{6}$$

$$\Box\Big(\|ks\| \implies \Box(\Diamond\|ks\|)\Big) \tag{7}$$

$$\Box(\|bb\|) \tag{8}$$

Fig. 5. Specification of activation constraints for blackboard architectures.

Activation Constraints. Activation constraints restrict the activation or deactivation of components. They are introduced by CDs and refined by activation assertions (AAs).

In our example, the "$bb\colon BB$" and "$ks\colon KS$" annotations for a BB and KS interface, respectively, denote the condition that there are *unique* BB and KS components denoted bb and ks, respectively. Moreover, we require three more activation constraints formulated as AAs in Fig. 5:

- By Eq. (6) we require ks to be active whenever bb posts a problem.
- By Eq. (7) we require a fairness condition for the activation of an already activated ks.
- By Eq. (8) we require that bb is always active.

Connection Constraints. Connection constraints restrict the connection between components. They are introduced by CDs and refined by connection assertions CAs.

In our example, connection constraints are also specified graphically by the CD in Fig. 4c. The *solid* connections between the ports denote a constraint requiring that the ports of a KS component are connected with the corresponding ports of a BB component as depicted, whenever both components are active.

Behavior Specifications. Behavior is specified in terms of BTAs as introduced in Sect. 3. Note that we do not consider activation and deactivation of a component when specifying its behavior. Rather, this is done in a separate specification and our calculus can then be used to reason about such behavior, in a dynamic environment as well.

In Fig. 6, we specify two simple properties for BB components. They merely require messages from their input ports to be forwarded to the corresponding output ports. Figure 7 provides a specification of the KS's behavior. The property requires that whenever a problem is received it is guaranteed to be eventually solved.

Spec BB_Bhv **uses** *Blackboard*

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

$$\Box\Big(p \in i_p \implies p \in p_p\Big) \quad (9)$$

$$\Box\Big(p \in p_s \implies p \in o_s\Big) \quad (10)$$

Fig. 6. Specification of behavior for blackboard components.

Spec KS_Bhv **uses** *Blackboard*

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

$$\Box\Big(p \in p_p \implies \Diamond\big(s(p) \in p_s\big)\ \Big) \quad (11)$$

Fig. 7. Specification of behavior for knowledgesource components.

5.2 Dynamic Blackboard Architectures: Verification

In the following, we demonstrate how the calculus proposed in Sect. 4 can be used to verify a simple property of blackboard architectures as specified above.

A simple property of a blackboard architecture as specified above is that a problem is always solved. Expressed in terms of a behavior assertion over a blackboard interface, it looks as follows:

$$\Box\Big(p \in i_p \implies \Diamond\big(s(p) \in o_s\big)\Big). \tag{12}$$

It actually resembles the behavior property of KS components. Its proof is split into 4 parts.

First, we apply introduction for the globally and eventually operators to our goal. Thereby we use Hilbert's ε-operator to denote an arbitrary but fixed element satisfying a certain property. Moreover, we use *Ass* to abbreviate the assumption $(t, t', n)\ {}^{t}_{k}{\models}_{(bb)} p \in i_p$ for later reference.

$$[n \geq 0]\ \dfrac{[Ass]\ \dfrac{\dfrac{\begin{array}{c}\vdots\\ (t, t', \varepsilon n'.\ n' \geq \langle bb \overset{n}{\vee} t\rangle)\ {}^{t}_{k}{\models}_{(bb)}(s(p) \in o_s)\end{array} \qquad \dfrac{}{\langle bb \overset{n}{\vee} t\rangle \leq \big(\varepsilon n'.\ n' \geq \langle bb \overset{n}{\vee} t\rangle\big)}}{(t, t', n)\ {}^{t}_{k}{\models}_{(bb)} \Diamond\big(s(p) \in o_s\big)}\ \text{EvtI}}{(t, t', n)\ {}^{t}_{k}{\models}_{(bb)}\Big(p \in i_p \implies \Diamond\big(s(p) \in o_s\big)\Big)}\ \text{ImpI}}{(t, t', 0)\ {}^{t}_{k}{\models}_{(bb)}\Box\Big(p \in i_p \implies \Diamond\big(s(p) \in o_s\big)\Big)}\ \text{GlobI}$$

We are now left with the goal of showing that the solution to the original problem p is provided by the blackboard at port o_s at some point in time later than the last activation of the blackboard. To discharge the proof obligation, we apply elimination for the globally operator and the behavior specification of blackboards. In the following, we abbreviate $\varepsilon n'.\ n' \geq \langle bb \overset{n}{\vee} t\rangle$ with n^*.

$$\dfrac{\dfrac{\dfrac{}{n^* \geq \langle c \overset{0}{\vee} t\rangle} \quad \dfrac{}{(t,t',0)\ {}^{t}_{k}{\models}_{(bb)} \Box\big(s(p) \in p_s \implies s(p) \in o_s\big)}\ \text{Eq. (10)}}{(t,t',n^*)\ {}^{t}_{k}{\models}_{(bb)} s(p) \in p_s \implies s(p) \in o_s}\ \text{GlobE} \qquad \begin{matrix}\vdots \\ (t,t',n^*)\ {}^{t}_{k}{\models}_{(bb)} (s(p) \in p_s)\end{matrix}}{(t,t',n^*)\ {}^{t}_{k}{\models}_{(bb)} (s(p) \in o_s)}\ \text{ImpE}$$

We are left with the goal of showing that the solution for p is indeed received by the blackboard. To this end, we apply connection constraints from the CD as well as elimination rules for eventually and globally.

$$\dfrac{\dfrac{\dfrac{\dfrac{\begin{matrix}\vdots \\ (t,t',n)\ {}^{t}_{k}{\models}_{(ks)} p \in p_p\end{matrix} \qquad \dfrac{\dfrac{}{(t,t',0)\ {}^{t}_{k}{\models}_{(ks)} \Box\Big(p \in p_p \implies \diamond\big(s(p) \in p_s\big)\Big)}\ \text{Eq. (11)} \ \dfrac{}{n \geq \langle c \overset{0}{\vee} t\rangle}}{(t,t',n)\ {}^{t}_{k}{\models}_{(ks)} \Big(p \in p_p \implies \diamond\big(s(p) \in p_s\big)\Big)}\ \text{GlobE}}{(t,t',n)\ {}^{t}_{k}{\models}_{(ks)} \diamond(s(p) \in p_s)}\ \text{ImpE}}{(t,t',\varepsilon n'.\ n' \geq \langle bb \overset{n}{\vee} t\rangle)\ {}^{t}_{k}{\models}_{(ks)} (s(p) \in p_s)}\ \text{EvtE}_{\text{a}}\text{, Eq. (7)}}{(t,t',\varepsilon n'.\ n' \geq \langle bb \overset{n}{\vee} t\rangle)\ {}^{t}_{k}{\models}_{(bb)} (s(p) \in p_s)}\ \text{Fig. 4c, Eq. (6)}$$

Finally it remains to show that the knowledgesource indeed receives the original problem. To discharge this obligation, we simply again apply the constraints induced by the CD as well as the behavioral specification of the blackboard component.

$$\dfrac{\dfrac{\dfrac{}{(t,t',n)\ {}^{t}_{k}{\models}_{(bb)} p \in i_p}\ \text{Ass} \quad \dfrac{\dfrac{}{(t,t',0)\ {}^{t}_{k}{\models}_{(bb)} \Box\Big(p \in i_p \implies \big(p \in p_p\big)\Big)}\ \text{Eq. (9)} \ \dfrac{}{n \geq \langle c \overset{0}{\vee} t\rangle}}{(t,t',n)\ {}^{t}_{k}{\models}_{(bb)} p \in i_p \implies p \in p_p}\ \text{GlobE}}{\dfrac{(t,t',n)\ {}^{t}_{k}{\models}_{(bb)} p \in p_p}{(t,t',n)\ {}^{t}_{k}{\models}_{(ks)} p \in p_p}\ \text{Fig. 4c, Eq. (8)}}\ \text{ImpE}$$

Note that one of the premises is closed by reference to the assumption *Ass* obtained at the beginning of the proof.

6 Related Work

Related work can be found in two different areas: work on the *specification* of *dynamic architectures* in general and *calculi* about *dynamic systems* specifically.

Over the last years, some approaches to the specification of dynamic architectures in general have emerged. One related approach comes from Le Métayer [12], who applies graph theory to specify architectural evolution. Similar to our work, the author proposed to model dynamic architectures as a sequence of graphs and to employ graph grammars as a technique to specify architectural evolution. A similar approach comes from Hirsch and Montanari [11], who employ hypergraphs as a formal model to represent styles and their reconfigurations. Another, closely related approach is the one used by Wermlinger et al. [20]. The authors combine behavior and structure to model dynamic reconfigurations. Recently, categorical approaches to dynamic architecture reconfiguration have appeared, such as the work of Castro et al. [7] and Fiadeiro and Lopes [9]. While they all introduce models for dynamic architectures similar to ours, they do not provide a calculus to reason about such architectures. Thus, we complement their work by providing rules to reason about such architectures.

A second area of work concerns approaches to reason about dynamic systems in general: Pioneering work in this area goes back to Milner in his well-known work on the π-calculus [17]. Here, the author provides a set of rules to reason about reconfigurable systems in general. The main idea behind the underlying model is that channels can be passed as messages between processes, which can then exchange messages over these channels. Another foundational model of dynamic systems which provides rules to reason about such systems is the Chemical Abstract Machine (CHAM) [1]. It is built upon the chemical metaphor and models a system as multi-set transformers. Thereby it also provides a set of general laws to reason about such systems. Finally, the ambient calculus [6] can be seen as an advancement of the CHAM. In contrast to membranes in CHAM, ambients provide stronger protection and provide mobility for sub-solutions as well. While all these approaches provide rules to reason about dynamic systems in general, their underlying model of dynamic systems is different from our model of dynamic architectures. Thus, we actually complement their work by providing rules to reason about different types of systems.

7 Conclusion

In this paper, we introduce a framework to reason about the behavior of components deployed to a dynamic environment. The major contributions of the paper can be summarized as follows: (i) We extend our model of dynamic architectures introduced in [15] with the notion of behavior traces to model behavior of single components. Thereby we also characterize an operator to extract the behavior of single components out of a given configuration trace. (ii) We introduce the notion of behavior trace assertions to specify behavior of single components and provide its formal semantics in terms of behavior traces. (iii) We provide a calculus to reason about the behavior of components in dynamic architectures. It is in a natural deduction style and provides introduction and elimination rules for each operator of behavior trace assertions. (iv) We show soundness and relative completeness of the calculus.

Our results can be used to support the verification of dynamic architectures. This was demonstrated by applying our calculus to verify a property for a dynamic version of the blackboard architecture pattern. Our overall research is directed towards a unified framework for the specification and verification of patterns for dynamic architectures. By introducing the calculus, we provide an important step towards this overall goal. However, future work is still required in three major directions: (i) To better support verification, we are aiming at integrating the calculus in Isabelle/HOL. Very much in the spirit of LCF [10], we are currently working on a mechanized proof of the rules of the calculus from first principles. (ii) Moreover, the calculus should be extended to better integrate port connections. (iii) We are currently looking for ways to leverage the hierarchical nature of patterns for their verification. Thus, we are interested in theoretical results of how results for one pattern can be reused for the verification of other, related patterns.

Acknowledgments. We would like to thank Manfred Broy, Mario Gleirscher, Vasileios Koutsoumpas, and the anonymous reviewers of ICTAC 2017 for their comments and helpful suggestions. The work was partially funded by the German Federal Ministry of Education and Research (BMBF) under grant "01Is16043A".

References

1. Berry, G., Boudol, G.: The chemical abstract machine. Theor. Comput. Sci. **96**(1), 217–248 (1992)
2. Bradbury, J.S., et al.: A survey of self-management in dynamic software architecture specifications. In: Proceedings of the 1st ACM SIGSOFT Workshop on Self-Managed Systems, pp. 28–33 (2004)
3. Broy, M.: A logical basis for component-oriented software and systems engineering. Comput. J. **53**(10), 1758–1782 (2010)
4. Broy, M.: A model of dynamic systems. In: Bensalem, S., Lakhneck, Y., Legay, A. (eds.) ETAPS 2014. LNCS, vol. 8415, pp. 39–53. Springer, Heidelberg (2014). doi:10.1007/978-3-642-54848-2_3
5. Buschmann, F., et al.: Pattern-Oriented Software Architecture: A System of Patterns (1996)
6. Cardelli, L., Gordon, A.D.: Mobile ambients. Theor. Comput. Sci. **240**(1), 177–213 (2000)
7. Castro, P.F., Aguirre, N.M., López Pombo, C.G., Maibaum, T.S.E.: Towards managing dynamic reconfiguration of software systems in a categorical setting. In: Cavalcanti, A., Deharbe, D., Gaudel, M.-C., Woodcock, J. (eds.) ICTAC 2010. LNCS, vol. 6255, pp. 306–321. Springer, Heidelberg (2010). doi:10.1007/978-3-642-14808-8_21
8. Clements, P.C.: A survey of architecture description languages. In: Proceedings of the 8th International Workshop on Software Specification and Design, p. 16 (1996)
9. Fiadeiro, J.L., Lopes, A.: A model for dynamic reconfiguration in serviceoriented architectures. Softw. Syst. Model. **12**(2), 349–367 (2013)
10. Gordon, M.J., Milner, A.J., Wadsworth, C.P.: Edinburgh LCF. LNCS, vol. 78. Springer, Heidelberg (1979). (Ed. by G. Goos and J. Hartmanis. 1st ed.)
11. Hirsch, D., Montanari, U.: Two graph-based techniques for software architecture reconfiguration. Electron. Notes Theor. Comput. Sci. **51**, 177–190 (2002)
12. Le Mtayer, D.: Describing software architecture styles using graph grammars. IEEE Trans. Softw. Eng. **24**(7), 521–533 (1998)
13. Manna, Z., Pnueli, A.: The Temporal Logic of Reactive and Concurrent Systems. Springer, New York (1992). doi:10.1007/978-1-4612-0931-7

14. Marmsoler, D.: On the specification of constraints for dynamic architectures. ArXiv e-prints, March 2017. arXiv: 1703.06823
15. Marmsoler, D., Gleirscher, M.: Specifying properties of dynamic architectures using configuration traces. In: Sampaio, A., Wang, F. (eds.) ICTAC 2016. LNCS, vol. 9965, pp. 235–254. Springer, Cham (2016). doi:10.1007/978-3-319-46750-4_14
16. Medvidovic, N.: ADLs and dynamic architecture changes. In: Joint Proceedings of the Second International Software Architecture Workshop and International Workshop on Multiple Perspectives in Software Development on SIGSOFT 1996 Workshops, pp. 24–27 (1996)
17. Milner, R.: Communicating and Mobile Systems: The π-calculus (1999)
18. Shaw, M., Garlan, D.: Software Architecture: Perspectives on an Emerging Discipline, vol. 1 (1996)
19. Taylor, R.N., Medvidovic, N., Dashofy, E.M.: Software Architecture: Foundations, Theory, and Practice. Wiley, Hoboken (2009)
20. Wermelinger, M., Lopes, A., Fiadeiro, J.L.: A graph based architectural (re)configuration language. Softw. Eng. Notes **26**(5), 21–32 (2001)
21. Wirsing, M.: Algebraic specification. In: van Leeuwen, J. (ed.) Handbook of Theoretical Computer Science, vol. B, Cambridge, pp. 675–788 (1990)