A Denotational Semantics for Dynamic Architectures

Diego Marmsoler, Technical University of Munich
Email: diego.marmsoler@tum.de

***Abstract*—With the emergence of mobile and adaptive computing, dynamic architectures have become increasingly important. In such architectures, components can appear and disappear, and connections between them can change over time. Verification of such architectures is performed over the composition of its components, which is usually defined in an operational style. Sometimes however, a denotational style might be more convenient for verification. Thus, in the following paper, we propose a denotational semantics for composition in dynamic architectures based on fixed points in lattices. We show that it is well-defined by proving that fixed points are guaranteed to exist. Finally, we use our definition to derive a logical characterization of composition, which forms the basis of a framework for the interactive verification of dynamic architectures.**

***Index Terms*—Dynamic Architectures; Denotational Semantics; Lattice Theory; Architecture Verification;**

I. Introduction

The architecture of a system describes its components and connections between them. Recent trends in computing, such as mobile and ubiquitous computing, require architectures to adapt dynamically: new components may join or leave the network and connections between them may change over time. Examples for such architectures range from well-known object-oriented applications over traditional P2P applications to more recent applications, such as Blockchain architectures [28].

Dynamic architectures are usually specified in three steps [16]. (i) First, the interfaces of the components are specified by means of architecture diagrams. Fig. 1, for example, depicts such a diagram for a dynamic Publisher-Subscriber architecture [7]. It consists of two types of components: Publisher components, which publish messages associated with events, and Subscriber components, which can join the architecture and subscribe to events to receive messages associated with these events. (ii) Component types are then specified by adding assertions about the behavior of components to the interfaces. A specification for Publisher components, which requires them to forward messages received on its input port i_1 to its output port o, for example, could look as follows:

$$\Box\big(i_1 = [e, m] \longrightarrow \bigcirc(o = [e, m])\big)$$

(iii) Finally, activation and deactivation of components of certain types and connections between them is specified by means of an architecture configuration. An architecture configuration specification for our Publisher-Subscriber example, which requires that there exists a *unique* Publisher component which is always active, could look as follows:

$$\Box\big(\wr p \wr \wedge \forall p' : \wr p' \wr \longrightarrow p' = p\big)$$

Here, $\wr p \wr$ is a predicate which denotes that component p of type $\mathit{Publisher}$ is active at the current point in time.

Specifications for dynamic architectures are verified by proving properties over the *composition* of its components according to the corresponding architecture configuration [16]. As of today, approaches for the verification of dynamic architectures focus on an operational style to semantics of composition [1], [3], [11], [24]. Sometimes, however, a denotational approach would be more convenient [27].

To close this gap, this paper proposes a denotational semantics for composition in dynamic architectures based on fixed points in lattices. To this end, we provide the following contributions:

1) We introduce a novel, semantic domain for dynamic architectures together with a corresponding ordering for elements of the domain and show that they form a complete lattice.
2) We introduce a function which calculates composition in dynamic architectures over our domain and show that it is monotonic w.r.t. the ordering introduced for that domain.
3) We define composition in dynamic architectures based on fixed points and ensure that it is well-defined by showing that such fixed points must always exist.
4) We use our definition to derive a law for composition in dynamic architectures, which we then use to justify soundness of our Isabelle-based [20] framework for the verification of dynamic architectures [15].

In the following, we first summarize our model for dynamic architectures. Then, we introduce the domain of prefix-closed architecture traces and define composition of dynamic architectures over this domain. Finally, we describe how our results can be used to justify soundness of our verification framework.

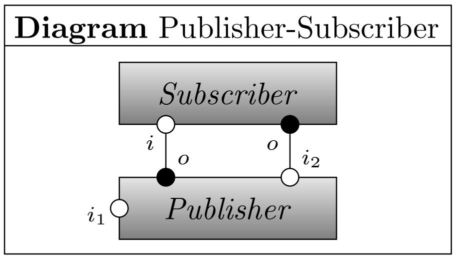


Figure 1. Architecture diagram for dynamic Publisher-Subscriber.

II. A Model of Dynamic Architectures

Our model for dynamic architectures [17], [18] is based on Broy's Focus theory [4]. It is completely formalized in Isabelle/HOL and available as entry `DynamicArchitectures` [12] of the Archive of Formal Proofs.

A. The Basics: Messages, Ports, and Interfaces

In our model, components communicate to each other by exchanging messages over ports. Thus, we assume the existence of set $\mathcal{M}$, containing all *messages*, and set $\mathcal{P}$, containing all *ports*, respectively. Moreover, we postulate the existence of a type function

$$\mathcal{T} \colon \mathcal{P} \to \wp(\mathcal{M}) \tag{1}$$

which assigns a set of messages to each port.

Ports are means to exchange messages between a component and its environment. This is achieved through the notion of port valuation. Roughly speaking, a valuation for a set of ports is an assignment of messages to each port.

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

$$\overline{P} \overset{\mathit{def}}{=} \Big\{\mu \in \big(P \to \wp(\mathcal{M})\big) \mid \forall p \in P \colon \mu(p) \subseteq \mathcal{T}(p)\Big\}$$

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, ..., respectively. For singleton sets we shall sometimes omit the set parentheses and simply write $[p_1, p_2, \ldots \mapsto m_1, m_2, \ldots]$.

In our model, ports may be valuated by *sets* of messages, meaning that a component can send/receive a set of messages via each of its ports at each point in time. A component may also send no message at all, in which case the corresponding port is valuated by the empty set.

The ports which a component may use to send and receive messages are grouped into so-called interfaces.

Definition 2 (Interface). *An* interface *is a pair (CI, CO), consisting of* disjoint *sets of* input ports *$CI \subseteq \mathcal{P}$ and* output ports *$CO \subseteq \mathcal{P}$. The set of all interfaces is denoted by $IF_{\mathcal{P}}$. For an interface $\mathit{if} = (CI, CO)$, we denote by*

- $\mathsf{in}(\mathit{if}) \overset{\mathit{def}}{=} CI$ *the set of input ports,*
- $\mathsf{out}(\mathit{if}) \overset{\mathit{def}}{=} CO$ *the set of output ports, and*
- $\mathsf{port}(\mathit{if}) \overset{\mathit{def}}{=} CI \cup CO$ *the set of all interface ports.*

B. Component Types

An important concept of our model are component types, i.e., interfaces with associated behavior.

In the following, we shall make use of finite as well as infinite *streams* [4]. Thereby, we denote with $(E)^*$ the set of all finite streams over elements of a given set E, by $(E)^\infty$ the set of all infinite streams over E, and by $(E)^\omega$ the set of all finite and infinite streams over E. The n-th element of a stream s is denoted with $s(n)$ and the first element is $s(0)$. Moreover, we shall use the following conventions for streams:

- With $\langle\rangle$ we denote the empty stream.
- With $e\&s$ we denote the stream resulting from appending stream s to element e.
- With $s\frown s'$ we denote the concatenation of stream s with stream s'.
- With $rg(s)$ we denote the set of all elements of a given stream s.
- With $\#s \in \mathbb{N}_\infty$ we denote the length of s.
- We use $s{\downarrow_n}$ to extract the first n (excluding the n-th) elements of a stream. Thereby $s{\downarrow_0} \overset{\text{def}}{=} \langle\rangle$.
- With $s' \sqsubseteq s$, we denote that s' is a prefix of s.
- With $bl(t)$ we denote the sequence t', such that $t = t' \frown \langle e \rangle$.

Roughly speaking, a component type is an interface with associated behavior. The behavior is given in terms of so-called behavior traces, streams of valuations of the corresponding interface ports.

Definition 3 (Component type). *A* component type *is a pair $(\mathit{if}, \mathit{bhv})$, consisting of*

- *an interface $\mathit{if} \in IF_{\mathcal{P}}$*
- *and a* non-empty *set of so-called* behavior traces $\mathit{bhv} \subseteq (\overline{\mathsf{port}(\mathit{if})})^\infty$

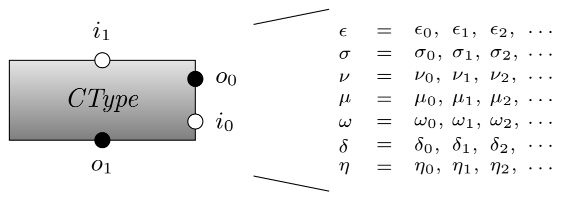
We shall use the same notation as introduced in Def. 2 to denote input, output, and all interface ports for component types. Moreover, for a component type $ct = (\mathit{if}, \mathit{bhv})$, we denote by

$$\mathsf{bhv}(ct) \overset{\mathit{def}}{=} \mathit{bhv} \tag{2}$$

the behavior of that type.

Example 1 (Component type). *Assuming $\mathcal{P}$ contains ports i_0, i_1, o_0, o_1. Figure 2 shows a conceptual representation of a component type $(\mathit{if}, \mathit{bhv})$, consisting of:*

- *Interface $\mathit{if} = (CI, CO)$, with*
  - *input ports $CI = \{i_0, i_1\}$, and*
  - *output ports $CO = \{o_0, o_1\}$.*
- *Behavior $\mathit{bhv} = \{\epsilon, \sigma, \nu, \mu, \omega, \delta, \eta\}$.*



i_1
o_0
i_0
o_1
CType
$\epsilon = \epsilon_0, \epsilon_1, \epsilon_2, \cdots$
$\sigma = \sigma_0, \sigma_1, \sigma_2, \cdots$
$\nu = \nu_0, \nu_1, \nu_2, \cdots$
$\mu = \mu_0, \mu_1, \mu_2, \cdots$
$\omega = \omega_0, \omega_1, \omega_2, \cdots$
$\delta = \delta_0, \delta_1, \delta_2, \cdots$
$\eta = \eta_0, \eta_1, \eta_2, \cdots$

Figure 2. Conceptual representation of a component type with behavior $\mathit{bhv} = \{\epsilon, \sigma, \nu, \mu, \omega, \delta, \eta\}$.

C. Architecture Configurations

Component types specify the interface and the allowed behavior for components. However, they do not say anything about the activation and deactivation of components or their interconnections. Thus, in the following, we introduce the concept of an architecture configuration to address these aspects.

1) Components: Component types can be instantiated to obtain components of that type. We shall use the same notation as introduced for component types in Def. 3, to access ports and behavior assigned to a component. Note, however, that instantiating a component leads to the notion of *component port*, which is a port combined with the corresponding component identifier. Thus, for a family of components $(\mathcal{C}_{ct})_{ct\in\mathcal{CT}}$ over a set of component types $\mathcal{CT} \subseteq CT_{\mathcal{I}}$, we denote by:

- $\mathsf{in}(\mathcal{C}) \overset{\text{def}}{=} \bigcup_{c\in\mathcal{C}} (\{c\} \times \mathsf{in}(c))$, the set of *component input ports*,
- $\mathsf{out}(\mathcal{C}) \overset{\text{def}}{=} \bigcup_{c\in\mathcal{C}} (\{c\} \times \mathsf{out}(c))$, the set of *component output ports*,
- $\mathsf{port}(\mathcal{C}) \overset{\text{def}}{=} \mathsf{in}(\mathcal{C}) \cup \mathsf{out}(\mathcal{C})$, the set of all *component ports*.

Moreover, we may lift the typing function (introduced for ports at the beginning of the chapter), to corresponding component ports:

$$\mathcal{T}((c,p)) \overset{\text{def}}{=} \mathcal{T}(p) \ .$$

Finally, we can generalize our notion of port valuation (Def. 1) for *component ports* $CP \subseteq \mathcal{C}\times\mathcal{P}$ to so-called *component port valuations*:

$$\overline{CP} \overset{\text{def}}{=} \left\{\mu \in \left(CP \to \wp(\mathcal{M})\right) \mid \forall cp \in CP\colon \mu(cp) \subseteq \mathcal{T}(cp)\right\}$$

To better distinguish between ports and component ports, in the following, we shall use p, q, pi, po, $\ldots$; to denote ports and cp, cq, ci, co, $\ldots$; to denote component ports.

2) Architecture Snapshots: An architecture is modeled as a sequence of snapshots of its state during execution. To this end, in the following, we introduce the notion of architecture snapshot. Such a snapshot consists of snapshots of currently active components, i.e. interfaces with its ports valuated with messages, and connections between the ports of these components.

Definition 4 (Architecture snapshot)**.** *An* architecture snapshot *is a triple (C', N, μ), consisting of:*

- *a set of components $C' \subseteq \mathcal{C}$,*
- *a connection $N\colon \mathsf{in}(C') \to \wp(\mathsf{out}(C'))$, such that*

$$\forall ci \in \mathsf{in}(C')\colon \bigcup_{co\in N(ci)} \mathcal{T}(co) \subseteq \mathcal{T}(ci) \qquad (3)$$

- *a component port valuation $\mu \in \overline{\mathsf{port}(C')}$.*

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:

$$\forall ci \in \mathsf{in}(C')\colon N(ci) \neq \emptyset \implies \mu(ci) = \bigcup_{co\in N(ci)} \mu(co) \quad (4)$$

Note that Eq. (3) *guarantees that Eq.* (4) *does not violate type restrictions. The set of all possible architecture snapshots is denoted by $AS^{\mathcal{C}}_{\mathcal{T}}$.*

For an architecture snapshot $as = (C', N, \mu) \in AS^{\mathcal{C}}_{\mathcal{T}}$, we denote by

- *$CMP_{as} \overset{def}{=} C'$ the set of active components and with $\wr c \wr_{as} \overset{def}{\iff} c \in C'$, that a component $c \in \mathcal{C}$ is* active *in as,*
- *$CN_{as} \overset{def}{=} N$, its connection, and*
- *$val_{as} \overset{def}{=} \mu$, the port valuation.*

Moreover, given a component $c \in C'$, we denote by

$$\mathsf{cmp}^c_{as} \in \overline{\mathsf{port}(\{c\})} \overset{def}{=} \Big(\lambda p \in \mathsf{port}(\{c\})\colon \mu(p)\Big) \qquad (5)$$

the valuation of the component's ports.

Note that cmp^c_{as} is well-defined iff $\wr c \wr_{as}$. Moreover, note that connection N is modeled as a set-valued function from component input ports to component output ports, meaning that:

1) input/output ports can be connected to several output/input ports, respectively, and
2) not every input/output port needs to be connected to an output/input port (in which case the connection returns the empty set).

Note that by Eq. (4), the valuation of an input port connected to many output ports is defined to be the *union* of all the valuations of the corresponding, connected output ports.

Example 2 (Architecture snapshot)**.** *Figure 3 shows a conceptual representation of an architecture snapshot (C', N, μ), consisting of:*

- *active components $C' = \{c_1, c_2, c_3\}$ with corresponding component types (c_3, for example, is of a type as described in Ex. 1);*
- *connection N, defined as follows:*
  - $N((c_2, i_1)) = \{(c_1, o_1)\}$,
  - $N((c_3, i_1)) = \{(c_1, o_2)\}$,
  - $N((c_2, i_2)) = \{(c_3, o_1)\}$, *and*
  - $N((c_1, i_0)) = N((c_2, i_0)) = N((c_3, i_0)) = \emptyset$*; and*
- *component port valuation* $[(c_1, o_0), (c_2, i_1), (c_3, o_1), \cdots \mapsto \mathtt{M}_3, \mathtt{M}_5, \mathtt{M}_3, \cdots]$.

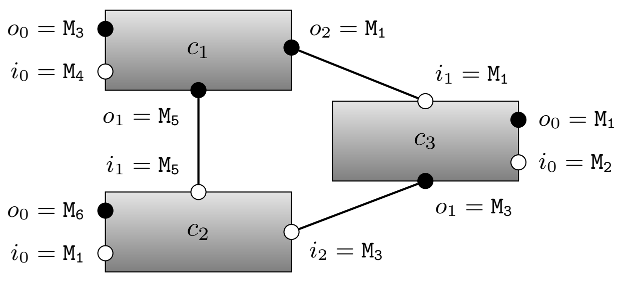


Figure 3. Architecture snapshot consisting of three components c_1, c_2, and c_3; a connection between ports (c_2, i_1) and (c_1, o_1), (c_2, i_2) and (c_3, o_1), and (c_3, i_1) and (c_1, o_2); and valuations of the component ports.

3) Architecture Traces: An *architecture trace* consists of a series of snapshots of an architecture during system execution. Thus, an architecture trace is modeled as a stream of architecture snapshots at certain points in time.

Definition 5 (Architecture trace)**.** *An* architecture trace *is a finite or infinite stream $t \in (AS^{\mathcal{C}}_{\mathcal{T}})^{\omega}$. For an architecture trace $t \in (AS^{\mathcal{C}}_{\mathcal{T}})^{\omega}$ and a component $c \in \mathcal{C}$, we denote with*

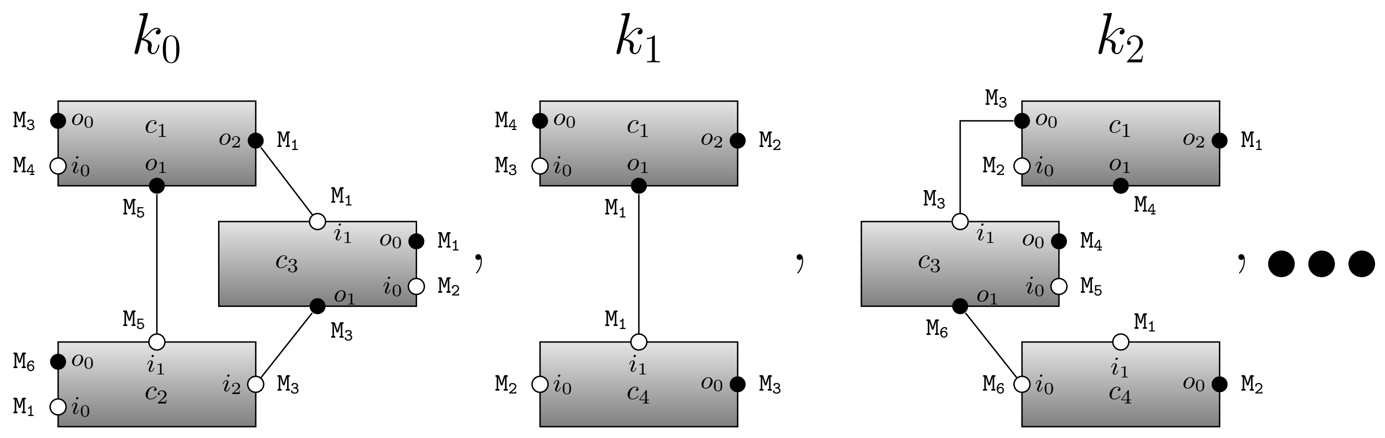


Figure 4. The first three architecture snapshots of an architecture trace.

- *$last(c,t)$, the* greatest $i \in \mathbb{N}$, *such that* $\wr c \wr_{t(i)}$,
- $c \overset{n}{\Leftarrow} t$, *the last time point less or equal to n at which c was* not *active in t, i.e., the* least $n' \in \mathbb{N}$, *such that* $n' = n \vee \left(n' < n \wedge \nexists n' \leq k < n \colon \wr c \wr_{t(k)}\right)$,
- $c \overset{n}{\leftarrow} t$, *the latest activation of component c (strictly) before n, and*
- $c \overset{n}{\rightarrow} t$, *the next point in time (after n) at which c is active in t.*

Note that our definition of architecture trace does not pose any restrictions on the elements in the sequence. Thus, an architecture's state can change arbitrarily from one point in time to the next.

Example 3 (Architecture trace)**.** *Figure 4 shows an architecture trace $t \in (AS_{\mathcal{T}}^{\mathcal{C}})^{\omega}$ with corresponding architecture snapshots $t(0) = k_0$, $t(1) = k_1$, and $t(2) = k_2$. Architecture snapshot k_0, for example, is described in Ex. 2.*

4) Architecture Configurations: Finally, we can define our notion of architecture configuration as a set of infinite architecture traces, satisfying certain properties.

Definition 6 (Architecture configuration)**.** *An* architecture configuration *is a* non-empty *set $\mathcal{A} \subseteq (AS_{\mathcal{T}}^{\mathcal{C}})^{\infty}$ of architecture traces, such that it does not restrict the behavior of components*[1].

Note that an architecture configuration does not restrict the behavior of components. A component's behavior, on the other hand is restricted in the specification of component types.

D. From Architecture Traces to Component Behavior

An important concept to connect architecture traces with component behavior is the notion of behavior projection. It is used to extract the behavior of a certain component out of a given architecture trace (Figure 5).

In the following, we provide a *co-recursive* definition for behavior projection.

[1]Since this restriction is not important for obtaining fixed points, we do not formalize it here.

Definition 7 (Behavior projection)**.** *Given an architecture trace $t \in (AS_{\mathcal{T}}^{\mathcal{C}})^{\omega}$. The* behavior projection *to component $c \in \mathcal{C}_{ct}$ of type $ct \in \mathcal{CT}$ is denoted by $\Pi_c(t) \in (\overline{\mathsf{port}(c)})^{\omega}$ and defined by the following equations:*

$$\begin{aligned} && \Pi_c(\langle\rangle) &= \langle\rangle \\ \wr c \wr_{as} &\implies & \Pi_c(as \mathbin{\&} t) &= \mathsf{cmp}_{as}^{c} \mathbin{\&} \Pi_c(t) \\ \neg\wr c \wr_{as} &\implies & \Pi_c(as \mathbin{\&} t) &= \Pi_c(t) \\ \left(\forall as \in rg(t) \colon \neg\wr c \wr_{as}\right) &\implies & \Pi_c(t) &= \langle\rangle \end{aligned}$$

Note that the structure of the equations provided in Def. 7 ensures productivity [10] and hence they resemble a valid *co-recursive* definition. Thus, projection is indeed well-defined by those equations.

Example 4 (Behavior projection)**.** *Applying behavior projection for component c_3 to the architecture trace described in Ex. 3 results in a behavior trace starting as follows:*

$$[i_0, i_1, o_0, o_1 \mapsto \mathtt{M}_2, \mathtt{M}_1, \mathtt{M}_1, \mathtt{M}_3],\ [i_0, i_1, o_0, o_1 \mapsto \mathtt{M}_5, \mathtt{M}_3, \mathtt{M}_4, \mathtt{M}_6],\ \cdots$$

III. Prefix-Closed Architectures

In the following, we introduce a novel domain for dynamic architectures, based on trace-based denotational semantics [8], [22].

Definition 8 (Prefix Closed Architecture)**.** *A set of architecture traces $A \subseteq (AS_{\mathcal{T}}^{\mathcal{C}})^{\omega}$ is called* prefix-closed*, whenever*

$$\forall t \in A, t' \in (AS_{\mathcal{T}}^{\mathcal{C}})^{\omega} \colon t' \sqsubseteq t \implies t' \in A \qquad (6)$$

Given a family of components $(\mathcal{C}_{ct})_{ct \in \mathcal{CT}}$ and an architecture configuration $\mathcal{A} \subseteq (AS_{\mathcal{T}}^{\mathcal{C}})^{\infty}$, we denote the set of all prefix-closed architectures by $[AS]_{\mathcal{T}}^{C}$.

Moreover, we introduce an ordering relation between architectures.

Definition 9 (Prefix Order for Architectures)**.** *Architectures can be ordered by the following prefix relation:*

$$A' \sqsubseteq A \overset{def}{\iff} \forall t' \in A' \; \exists t \in A \colon t' \sqsubseteq t$$

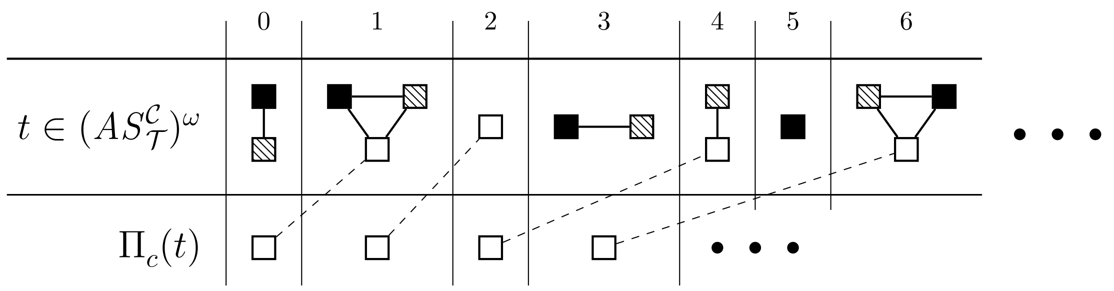


Figure 5. The above row shows a conceptual representation of an architecture trace t, which contains a component c, represented by an empty rectangle. The bottom row shows the result of projecting to the behavior of c in architecture trace t.

It is a well-known fact, that the prefix order forms a partial order over prefix-closed sets.

Lemma 1 (Partial Order). *$([AS]^{\mathcal{C}}_{\mathcal{T}}, \sqsubseteq)$ forms a partial order.*

Note that prefix closure is an important requirement to ensure that $\sqsubseteq$ is indeed a partial order over sets of architecture traces. For, as the following example shows, it is needed to ensure anti-symmetry of $\sqsubseteq$.

Example 5 (Why we need prefix closure). *Assume a set $A = \{\langle as \rangle\}$ and set $B = \{\langle as \rangle, \langle\rangle\}$ for an arbitrary architecture snapshot $as \in AS^{\mathcal{C}}_{\mathcal{T}}$. Since $\langle as \rangle \sqsubseteq \langle as \rangle$, it follows from Def. 9, that $A \sqsubseteq B$. Moreover, since $\langle\rangle \sqsubseteq \langle as \rangle$ and $\langle as \rangle \sqsubseteq \langle as \rangle$, according to Def. 9, $B \sqsubseteq A$. Thus, $A \sqsubseteq B$ and $B \sqsubseteq A$ though $A \neq B$ which shows that $\sqsubseteq$ is not anti-symmetric.* □

Similarly, it is well-known, that for prefix-closed sets, the prefix order is equal to the subset relation.

Lemma 2. *Let A and B be two* prefix-closed *sets of architecture traces. Then,*

$$A \sqsubseteq B \iff A \subseteq B \tag{7}$$

This property becomes very useful later on to construct least upper bounds for prefix-closed architectures.

Not only is the prefix order a partial order over prefix-closed architectures, but it is indeed a complete lattice w.r.t. $\sqsubseteq$.

Lemma 3. *The sets of* prefix-closed *sets of architecture traces form a complete lattice w.r.t. $\sqsubseteq$.*

IV. Composition for Dynamic Architectures

In the following section, we define composition of component types, according to a corresponding architecture configuration. Intuitively, the composition should return a set of all the *infinite* architecture traces which satisfy the architecture configuration and comply with the specification of component types.

A. Architecture Execution

We first introduce an auxiliary construct which models the change of the architecture for a single execution step.

Definition 10. *Let $(\mathcal{C}_{ct})_{ct \in \mathcal{CT}}$ be a family of components and $\mathcal{A} \subseteq (AS^{\mathcal{C}}_{\mathcal{T}})^{\infty}$ an architecture configuration. $\Gamma^{\mathcal{C}}_{\mathcal{T}} \colon [AS]^{\mathcal{C}}_{\mathcal{T}} \to \wp\big((AS^{\mathcal{C}}_{\mathcal{T}})^{\omega}\big)$ is defined as follows:*

$$\Gamma^{\mathcal{C}}_{\mathcal{T}}(T) = \Big\{ t \in (AS^{\mathcal{C}}_{\mathcal{T}})^{\omega} \mid t = \langle\rangle \vee \tag{8}$$

$$bl(t) \in T \tag{9}$$

$$\wedge \big(\exists t' \in \mathcal{A} \colon t \sqsubseteq t'\big) \tag{10}$$

$$\wedge \big(\forall ct \in CT,\ c \in C_{ct}\ \exists t' \in \mathsf{bhv}(ct) \colon \Pi_c(t) \sqsubseteq t'\big) \Big\} \tag{11}$$

The auxiliary construct Γ takes a prefix-closed set of architecture traces $T \in [AS]^{\mathcal{C}}_{\mathcal{T}}$, which is assumed to contain all histories of all possible computations so far. Then, it appends one additional *architecture snapshot* to each history (Eq. (9)). To ensure that the obtained trace is indeed a feasible computation according to the specifications, Eq. (10) checks its compatibility with the architecture configuration, and Eq. (11) its compatibility with the specification of component types.

As the following proposition tells us, the construct defined in Def. 10 preserves prefix closure for architecture traces.

Proposition 1. *Let $(\mathcal{C}_{ct})_{ct \in \mathcal{CT}}$ be a family of components and $\mathcal{A} \subseteq (AS^{\mathcal{C}}_{\mathcal{T}})^{\infty}$ an architecture configuration. For every prefix-closed set of architecture traces $T \in [AS]^{\mathcal{C}}_{\mathcal{T}}$, $\Gamma^{\mathcal{C}}_{\mathcal{T}}(T)$ is again a prefix-closed set of architecture traces.*

B. Fixed Points

We would like to take a fixed point of the above construct as the definition of composition. Thus, we first need to ensure that fixed points indeed exist. To this end, we first show that Γ is monotonic w.r.t. the ordering introduced in the last section.

Lemma 4. *For a family of components $(\mathcal{C}_{ct})_{ct \in \mathcal{CT}}$ and an architecture configuration $\mathcal{A} \subseteq (AS^{\mathcal{C}}_{\mathcal{T}})^{\infty}$, $\Gamma^{\mathcal{C}}_{\mathcal{T}} \colon [AS]^{\mathcal{C}}_{\mathcal{T}} \to [AS]^{\mathcal{C}}_{\mathcal{T}}$ as defined in Def. 10 is monotonic w.r.t. the ordering defined in Def. 9.*

Proof. Let $A, B \subseteq (AS^{\mathcal{C}}_{\mathcal{T}})^{\omega}$ and assume $A \sqsubseteq B$. We show $\forall t' \in \Gamma^{\mathcal{C}}_{\mathcal{T}}(A)\ \exists t \in \Gamma^{\mathcal{C}}_{\mathcal{T}}(B) \colon t' \sqsubseteq t$ to conclude $\Gamma^{\mathcal{C}}_{\mathcal{T}}(A) \sqsubseteq \Gamma^{\mathcal{C}}_{\mathcal{T}}(B)$ according to Def 9: To this end, let $t' \in \Gamma^{\mathcal{C}}_{\mathcal{T}}(A)$. According to Def. 10, $bl(t) \in A$ and $\exists t' \in \mathcal{A} \colon t \sqsubseteq t'$ and $\forall ct \in CT,\ c \in C_{ct}\ \exists t' \in \mathsf{bhv}(ct) \colon \Pi_c(t) \sqsubseteq t'$. Since $A \sqsubseteq B$, according to Prop. 2, we have $A \subseteq B$ and since $bl(t) \in A$ we conclude $bl(t) \in B$. Moreover, since $\exists t' \in \mathcal{A} \colon t \sqsubseteq t'$ and $\forall ct \in CT,\ c \in C_{ct}\ \exists t' \in \mathsf{bhv}(ct) \colon \Pi_c(t) \sqsubseteq t'$ conclude $t \in \Gamma^{\mathcal{C}}_{\mathcal{T}}(B)$ by Def. 10. □

Now, we can simply apply Tarski's fixed point theorem [26], which guarantees existence of least and greatest fixed points.

Theorem 1. *For a family of components $(\mathcal{C}_{ct})_{ct\in\mathcal{CT}}$ and an architecture configuration $\mathcal{A} \subseteq (AS^{\mathcal{C}}_{\mathcal{T}})^{\infty}$, $\Gamma^{\mathcal{C}}_{\mathcal{T}}\colon [AS]^{\mathcal{C}}_{\mathcal{T}} \to [AS]^{\mathcal{C}}_{\mathcal{T}}$ as defined in Def. 10 has a least and greatest fixed point.*

Proof. From Lem. 3 and Lem. 4 by Tarski [26]. □

C. Composition

The final question which remains, is whether to take the least or the greatest fixed point to define composition. As mentioned in the introduction of this section, we want all *infinite* architecture traces, which satisfy an architecture configuration specification and comply with the specifications of each component type. As the following proposition shows, this is not the case for the least fixed point.

Proposition 2. *Let a $(\mathcal{C}_{ct})_{ct\in\mathcal{CT}}$ be a family of components and $\mathcal{A} \subseteq (AS^{\mathcal{C}}_{\mathcal{T}})^{\infty}$ an architecture configuration. The least fixed point of $\Gamma^{\mathcal{C}}_{\mathcal{T}}$ as defined in Def. 10 is given by the set of all* finite *architecture traces $t \in (AS^{\mathcal{C}}_{\mathcal{T}})^{*}$, such that*

$$\exists t' \in \mathcal{A}\colon t \sqsubseteq t' \land \tag{12}$$

$$\forall ct \in CT,\ c \in C_{ct}\ \exists t' \in \mathsf{bhv}(ct)\colon \Pi_c(t) \sqsubseteq t' \tag{13}$$

Thus, the least fixed point actually contains all *finite* architecture traces, which comply with the specification of the architecture itself (Eq. (12)) and with the specification of the behavior of each component (Eq. (13)).

Thus, we take the greatest fixed point *fix* to define composition for dynamic architectures.

Definition 11 (Composition)**.** Composition *of a family of components $(\mathcal{C}_{ct})_{ct\in\mathcal{CT}}$ according to an architecture configuration $\mathcal{A} \subseteq (AS^{\mathcal{C}}_{\mathcal{T}})^{\infty}$ is defined as follows:*

$$\bigotimes_{\mathcal{A}}(\mathcal{C}) \quad \overset{def}{=} \quad \mathit{fix}(\Gamma^{\mathcal{C}}_{\mathcal{T}}) \cap (AS^{\mathcal{C}}_{\mathcal{T}})^{\infty}$$

where $\Gamma^{\mathcal{C}}_{\mathcal{T}}$ is defined as in Def. 10.

Note that, according to Thm. 1, the greatest fixed point always exists, which is why Def. 11 is always well-defined. Moreover, note that the greatest fixed point actually contains all *finite* and *infinite* architecture traces which satisfy our conditions. However, since we are only interested in the *infinite* ones, we take the intersection of the fixed point with all infinite architecture traces to define our notion of composition.

We conclude this section with a result which shows that Def. 11 indeed satisfies our intuition of composition as described in the introduction of this section.

Theorem 2. *Let $(\mathcal{C}_{ct})_{ct\in\mathcal{CT}}$ be a family of components and $\mathcal{A} \subseteq (AS^{\mathcal{C}}_{\mathcal{T}})^{\infty}$ an architecture configuration. Then,*

$$\bigotimes_{\mathcal{A}}(\mathcal{C}) = \big\{t \in \mathcal{A} \mid \\ \forall ct \in CT,\ c \in C_{ct}\ \exists t' \in \mathsf{bhv}(ct)\colon \Pi_c(t) \sqsubseteq t'\big\}$$

where $\Gamma^{\mathcal{C}}_{\mathcal{T}}$ is defined as in Def. 11.

Proof. The greatest fixed point of $\Gamma^{\mathcal{C}}_{\mathcal{T}}$ is given by

$$\mathit{fix}(\Gamma^{\mathcal{C}}_{\mathcal{T}}) = \Big\{t \in (AS^{\mathcal{C}}_{\mathcal{T}})^{\omega} \ \Big|\ \big(\exists t' \in \mathcal{A}\colon t \sqsubseteq t'\big) \land \\ \big(\forall ct \in CT,\ c \in C_{ct}\ \exists t' \in \mathsf{bhv}(ct)\colon \Pi_c(t) \sqsubseteq t'\big)\Big\}$$

Restricting this to only infinite traces yields

$$\big\{t \in \mathcal{A} \mid \forall ct \in CT,\ c \in C_{ct}\ \exists t' \in \mathsf{bhv}(ct)\colon \Pi_c(t) \sqsubseteq t'\big\}\square$$

V. Verifying Dynamic Architectures in Isabelle

In previous work [14], [15], we introduced a framework for the interactive verification of dynamic architectures, based on the interactive theorem prover Isabelle [20]. Figure 6 shows how a dynamic architecture can be verified using the framework: First, the architecture is specified in terms of component behavior and an architecture configuration. Both specifications are then interpreted over a formalization of the model of architecture traces introduced in Sect. II. However, while the behavior specification is interpreted using the formalization of component projection, the specification of the architecture configuration is directly interpreted over architecture traces.

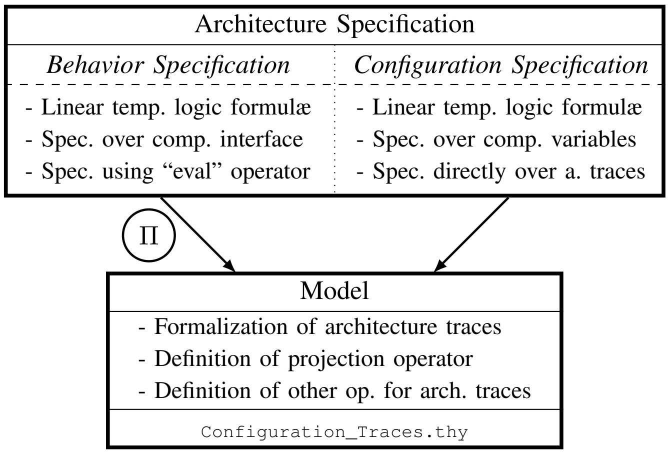

Architecture Specification
Behavior Specification
Configuration Specification
- Linear temp. logic formulæ
- Spec. over comp. interface
- Spec. using "eval" operator
- Linear temp. logic formulæ
- Spec. over comp. variables
- Spec. directly over a. traces
Π
Model
- Formalization of architecture traces
- Definition of projection operator
- Definition of other op. for arch. traces
Configuration_Traces.thy

Figure 6. Overview of the Interactive Architecture Verification framework.

In the following, we apply the results presented in this paper to demonstrate soundness of the framework. Thus, we first summarize some key aspects of the framework and then we explain how Thm. 2 can be used to justify its soundness.

A. Formalizing Dynamic Architectures

The model presented in Sect. II is implemented by theory `Configuration_Traces` [12]. In Sect. "Specifying Dynamic Architectures" it provides a formalization of the notion of architecture traces:

typedecl *cnf*
type-synonym *trace* = *nat* ⇒ *cnf*
consts *arch*:: *trace set*

According to Def. 6, we formalized an architecture as a set of traces *arch*. Architecture traces, on the other hand, are formalized as functions from *nat* to an abstract type *cnf*, which is a type containing all possible architecture snapshots (Def. 4). In addition, theory `Configuration_Traces` contains formalizations of many important concepts for architecture traces discussed in Sect. II, such as component activation (described in Def. 4), component projection (Def. 7), number of activations, and last activation (both described in Def. 5).

B. Specifying Dynamic Architectures

As shown in Fig. 6, an architecture is specified in two steps. First, an architecture configuration AS is specified in terms of so called architecture trace assertions, i.e. first-order linear temporal logic formulæ over architecture traces. They are formalized by means of Isabelle type synonyms:

```
type-synonym cta = trace ⇒ nat ⇒ bool
```

Then, component behavior $\mathtt{CS}_{ct}$ is specified for different types of components $ct \in CT$ by means of so called behavior trace assertions [13], i.e. first-order linear temporal logic formulæ over behavior traces. Similar to architecture trace assertions, they are implemented by means of Isabelle type synonyms:

```
type-synonym 'cmp bta = (nat ⇒ 'cmp) ⇒ nat ⇒ bool
```

This time, however, they are not interpreted over an architecture trace. Rather, they are interpreted over a behavior trace $nat \Rightarrow {}'cmp$. To interpret them over an architecture trace, we provide an evaluation operator $eval$:

```
definition eval:: 'id ⇒ (nat ⇒ cnf) ⇒ (nat ⇒ 'cmp) ⇒ nat
 ⇒ 'cmp bta ⇒ bool
 where eval cid t t' n γ ≡
 (∃ i≥n. ≀cid≀_t i) ∧
  γ (lnth ((π_cid(inf-llist t)) @_l (inf-llist t')))
   (the-enat (⟨cid #_n inf-llist t⟩)) ∨
 (∃ i. ≀cid≀_t i) ∧ (∄ i'. i'≥n∧≀cid≀_t i') ∧
  γ(lnth ((π_cid(inf-llist t)) @_l (inf-llist t')))(_cid↓t(n)) ∨
 (∄ i. ≀cid≀_t i) ∧ γ (lnth ((π_cid(inf-llist t)) @_l (inf-llist t'))) n
```

Roughly speaking, the operator evaluates a component type specification over an architecture trace using component projection.

Example 6 (Specifying Publisher-Subscriber architectures)**.** *Figure 7 depicts the specification of the Publisher-Subscriber example introduced in Sect. I in Isabelle. Note how operator "eval" is used in "bhvpb" to formalize a Publisher's behavior.*

```
locale publisher-subscriber =
  pb: dynamic-component pbcmp pbactive +
  sb: dynamic-component sbcmp sbactive +
  fixes pbi1 :: 'PB ⇒ ('evt × 'msg) set
    and pbi2 :: 'PB ⇒ ('evt set) subscription set
    and pbo :: 'PB ⇒ ('evt × 'msg)
    and sbi :: 'SB ⇒ ('evt × 'msg) set
    and sbo :: 'SB ⇒ ('evt set) subscription
  assumes bhvpb: ⋀t t' pId e m. ⟦t ∈ arch⟧ ⟹ pb.eval pId t t' 0
 (□_b ([λpb. (e,m) ∈ pbi1 pb]_b ⟶^b (○_b [λpb. pbo pb = (e,m)]_b)))
  and pbunique: ⋀t pId::'pid. ⟦t ∈ arch⟧ ⟹
  (□_c (ca (λas. ≀pId≀_as)  ∧^c
    (∀ _c pId'. (ca (λas. ≀pId≀_as) ⟶^c ca (λas. pId'=pId))))) t 0
```

Figure 7. Specification of Publisher-Subscriber architecture.

C. Verifying Dynamic Architectures

Verification is finally done over the conjunction of the architecture configuration specification and the evaluation of component type specifications. Thus, to verify a property P, we actually show

$$\bigwedge_{ct \in CT} \big(eval(\mathtt{CS}_{ct})\big) \;\wedge\; \mathtt{AS} \quad\Longrightarrow\quad \mathtt{P} \tag{14}$$

When verifying a specification using the framework, component behavior specifications are interpreted over architecture traces (using operator $eval$) and they are used to refine a given architecture configuration. Thus, to ensure soundness of the implementation, we must ensure that an architecture satisfies a specification $(\mathtt{AS}, \mathtt{CT}_{ct})$ iff the architecture corresponds to the composition of a set of component types satisfying CT according to an architecture configuration satisfying AS.

In the following, we use Thm. 2 to show this property and thus, justify soundness of the framework.

Proposition 3. *Let* AS *be an architecture configuration specification and* $\mathtt{CS}_{ct\in CT}$ *be a specification of component behavior. A family of components* $(\mathcal{C}_{ct})_{ct\in\mathcal{CT}}$ *fulfills* CS *and an architecture configuration* $\mathcal{A} \subseteq (AS^{\mathcal{C}}_{\mathcal{T}})^{\infty}$ *fulfills* AS *iff*

$$\bigotimes\nolimits_{\mathcal{A}}(\mathcal{C}) = \big\{t \in (AS^{\mathcal{C}}_{\mathcal{T}})^{\infty} \mid t \models \mathtt{AS} \;\wedge \\ \forall ct \in CT,\ c \in C_{ct}\ \exists t' \in \mathsf{bhv}(ct)\colon \Pi_c(t)^\frown t' \models \mathtt{CS}_{ct}\big\}$$

Proof. The forward direction can be justified as follows: Assume that a family of components $(\mathcal{C}_{ct})_{ct\in\mathcal{CT}}$ fulfills CS and an architecture configuration $\mathcal{A} \subseteq (AS^{\mathcal{C}}_{\mathcal{T}})^{\infty}$ fulfills AS. By Thm. 2, $\bigotimes_{\mathcal{A}}(\mathcal{C}) = \big\{t \in \mathcal{A} \mid \forall ct \in CT,\ c \in C_{ct}\ \exists t' \in \mathsf{bhv}(ct)\colon \Pi_c(t) \sqsubseteq t'\big\}$. Thus, since $t \models \mathtt{AS}$ and $\forall ct \in CT\colon ct \models \mathtt{CS}_{ct}$, conclude $\bigotimes_{\mathcal{A}}(\mathcal{C}) = \big\{t \in (AS^{\mathcal{C}}_{\mathcal{T}})^{\infty} \mid t \models \mathtt{AS} \wedge \forall ct \in CT,\ c \in C_{ct}\ \exists t' \in \mathsf{bhv}(ct)\colon \Pi_c(t)^\frown t' \models \mathtt{CS}_{ct}\big\}$. A symmetric argument can be used to justify the backward direction. ☐

VI. Related Work

Over the last years, several approaches emerged to support the formal specification of dynamic architectures which provide a good source for related work.

Two early approaches in this area where Darwin [11] and Wright [1]. Former is based on Milner's Π-Calculus [19] and latter on CSP [9]. More recently, Bozga et al. [3] extend the BIP component model [2]. Finally, Sanches et al. [24] extend the Archery language [23] to cope with dynamic architectures. All these approaches provide *operational* semantics for composition in dynamic architectures. While this is close to the implementation of such systems, for verification it is sometimes helpful to also have a denotational semantics. Thus, although inspired by the above works, the results presented in this paper complement them by providing denotational semantics for composition in dynamic architectures.

One exception to the above works is Broy's [5] work on dynamic Focus [6] in which he provides a denotational semantics for composition in dynamic architectures. However, unlike our work, the semantics is given by a logical characterization of composition, similar to the one presented in

Thm. 2. Thus, by providing semantics in terms of fixed points, we provide a constructive alternative to the above work. *In summary, to the best of our knowledge, this is the first work presenting a fixed-point based semantics for composition in dynamic architectures.*

VII. Conclusion

With this paper, we propose a denotational semantics for composition in dynamic architectures based on fixed points in lattices. To this end, we first introduce prefix-closed architectures (Def. 8) and an ordering between them (Def. 8), and show that they form a complete lattice (Lem. 3). Then, we define composition for dynamic architectures as all infinite architecture traces of the greatest fixed point of a function which extends a current execution (Def. 11). As a major result, we show that greatest fixed points are guaranteed to exist which ensures soundness of the definition (Thm. 1).

Our results have two major implications: First, they can be used to justify soundness of our framework for the verification of dynamic architectures. To this end, we derived a characteristic law for composition in dynamic architectures from our definition (Thm. 2) and use it to show soundness of our framework for the verification of dynamic architectures (Prop. 3). In addition, since the new definition is based on fixed points, we can use several results from fixed point theory to support the verification of dynamic architectures. For example, we can use Park induction for greatest fixed points [21] to reason about composition for dynamic architectures.

The definition of composition in dynamic architectures presented in this paper is based on *greatest fixed points* in lattices. For future work, we would like to come up with an alternative definition based on *least fixed points* in complete partial orders. This would allow us to use additional techniques for the verification of dynamic architectures, such as Park induction for least fixed points [21] or Scott's fixed-point induction [25].

Acknowledgments: I would like to thank *Manfred Broy* for useful discussions of composition in static architectures and *Alexander Knapp* for the hint to require prefix-closure for architecture traces. Moreover, I am grateful to *Fabio Mogavero* for the interesting discussion about fixed points in CPOs. Parts of the work on which we report in this paper was funded by the German Federal Ministry of Education and Research (BMBF) under grant no. 01Is16043A.

References

[1] R. Allen, R. Douence, and D. Garlan. Specifying and analyzing dynamic software architectures. In E. Astesiano, editor, *Fundamental Approaches to Software Engineering*, volume 1382 of *Lecture Notes in Computer Science*, pages 21–37. Springer Berlin Heidelberg, 1998.

[2] A. Basu, B. Bensalem, M. Bozga, J. Combaz, M. Jaber, T. Nguyen, and J. Sifakis. Rigorous component-based system design using the bip framework. *IEEE Software*, 28(3):41–48, May 2011.

[3] M. Bozga, M. Jaber, N. Maris, and J. Sifakis. Modeling dynamic architectures using dy-bip. In *Proceedings of the 11th International Conference on Software Composition*, SC'12, pages 1–16, Berlin, Heidelberg, 2012. Springer-Verlag.

[4] M. Broy. A logical basis for component-oriented software and systems engineering. *The Computer Journal*, 53(10):1758–1782, Feb. 2010.

[5] M. Broy. A model of dynamic systems. In S. Bensalem, Y. Lakhneck, and A. Legay, editors, *From Programs to Systems. The Systems Perspective in Computing*, volume 8415 of *Lecture Notes in Computer Science*, pages 39–53. Springer Berlin Heidelberg, 2014.

[6] M. Broy and K. Stolen. *Specification and development of interactive systems: focus on streams, interfaces, and refinement*. Springer Science & Business Media, 2012.

[7] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal. *Pattern-Oriented Software Architecture: A System of Patterns*. Wiley West Sussex, England, 1996.

[8] N. Francez, C. Hoare, D. J. Lehmann, and W. P. De Roever. Semantics of nondeterminism, concurrency, and communication. *Journal of Computer and System Sciences*, 19(3):290–308, 1979.

[9] C. A. R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, 1978.

[10] B. Jacobs and J. Rutten. A tutorial on (co)algebras and (co)induction. *EATCS Bulletin*, 62:62–222, 1997.

[11] J. Magee and J. Kramer. Dynamic structure in software architectures. In D. Garlan, editor, *SIGSOFT '96, Proceedings of the Fourth ACM SIGSOFT Symposium on Foundations of Software Engineering, San Francisco, California, USA, October 16-18, 1996*, pages 3–14. ACM, 1996.

[12] D. Marmsoler. Dynamic architectures. *Archive of Formal Proofs*, pages 1–65, July 2017. Formal proof development.

[13] D. Marmsoler. On the semantics of temporal specifications of component-behavior for dynamic architectures. In *Eleventh International Symposium on Theoretical Aspects of Software Engineering*. Springer, 2017.

[14] D. Marmsoler. Towards a calculus for dynamic architectures. In D. V. Hung and D. Kapur, editors, *Theoretical Aspects of Computing - ICTAC 2017 - 14th International Colloquium, Hanoi, Vietnam, October 23-27, 2017, Proceedings*, volume 10580 of *Lecture Notes in Computer Science*, pages 79–99. Springer, 2017.

[15] D. Marmsoler. A framework for interactive verification of architectural design patterns in isabelle/hol. In *International Conference on Formal Engineering Methods*, pages 251–269. Springer, 2018.

[16] D. Marmsoler. Hierarchical specication and verication of architecture design patterns. In *Fundamental Approaches to Software Engineering - 21th International Conference, FASE 2018, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2018, Thessaloniki, Greece, April 14-20, 2018, Proceedings*, 2018.

[17] D. Marmsoler and M. Gleirscher. On activation, connection, and behavior in dynamic architectures. *Scientific Annals of Computer Science*, 26(2):187–248, 2016.

[18] D. Marmsoler and M. Gleirscher. Specifying properties of dynamic architectures using configuration traces. In *International Colloquium on Theoretical Aspects of Computing*, pages 235–254. Springer, 2016.

[19] R. Milner. *Communicating and Mobile Systems: the π-calculus*. Cambridge university press, 1999.

[20] T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.

[21] D. Park. Fixpoint induction and proofs of program properties. *Machine intelligence*, 5, 1969.

[22] A. Roscoe. *The Theory and Practice of Concurrency*. Prentice Hall series in computer science. Prentice Hall, 1998.

[23] A. Sanchez, L. S. Barbosa, and D. Riesco. Bigraphical modelling of architectural patterns. In F. Arbab and P. C. Ölveczky, editors, *Formal Aspects of Component Software*, pages 313–330, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[24] A. Sanchez, A. Madeira, and L. S. Barbosa. On the verification of architectural reconfigurations. *Computer Languages, Systems & Structures*, 44:218–237, 2015.

[25] D. S. Scott. Lectures on a mathematical theory of computation. In *Theoretical Foundations of Programming Methodology*, pages 145–292. Springer, 1982.

[26] A. Tarski. A lattice-theoretical fixpoint theorem and its applications. *Pacific journal of Mathematics*, 5(2):285–309, 1955.

[27] G. Winskel. *The formal semantics of programming languages: an introduction*. MIT press, 1993.

[28] X. Xu, I. Weber, M. Staples, L. Zhu, J. Bosch, L. Bass, C. Pautasso, and P. Rimba. A taxonomy of blockchain-based systems for architecture design. In *2017 IEEE International Conference on Software Architecture, ICSA 2017, Gothenburg, Sweden, April 3-7, 2017*, 2017.