This is a post-peer-review, pre-copyedit version of an article published in Formal Aspects
of Computing. The final authenticated version is available online at: ...

Interactive Verification of Architectural Design Patterns in FACTum

Diego Marmsoler and Habtom Kashay Gidey
Technical University of Munich,
Germany

Abstract. Architectural design patterns (ADPs) are architectural solutions to common architectural design problems. They are an important concept in software architectures used for the design and analysis of architectures. An ADP usually constrains the design of an architecture and, in turn, guarantees some desired properties for architectures implementing it. Sometimes, however, the constraints imposed by an ADP do not lead to the claimed guarantee. Thus, applying such patterns for the design of architectures might result in architectures which do not fulfill their intended requirements. To address this problem, we propose an approach for the verification of ADPs, based on interactive theorem proving. To this end, we introduce a model for dynamic architectures and a language for the specification of ADPs over this model. Moreover, we propose a framework for the interactive verification of such specifications based on Isabelle/HOL. In addition we describe an algorithm to map a specification to a corresponding Isabelle/HOL theory over our framework. To evaluate the approach, we implement it in Eclipse/EMF and use it for the verification of four ADPs: variants of the Singleton, the Publisher-Subscriber, the Blackboard pattern, and a pattern for Blockchain architectures. With our approach we complement traditional approaches for the verification of architectures, which are usually based on automatic verification techniques such as model checking.

Keywords: Architecture Design Patterns; Interactive Theorem Proving; Architecture Verification; FACTum; Algebraic Specification; Isabelle

1. Introduction

The architecture of a system describes the overall organization of a system into components and connections between these components [BS01]. Since software systems are becoming increasingly big and complex, the architecture of a system plays an ever more important role in their development. Architectural design patterns (ADPs) are an important tool in software engineering employed for the conceptualization and analysis of architectures. They capture design experience and are regarded as the “Grand Tool” for designing a software

Correspondence and offprint requests to: Diego Marmsoler, Technische Universität München, Institut für Informatik – Lehrstuhl IV (I4), Boltzmannstr. 3, 85748 Garching bei München, Germany. e-mail: diego.marmsoler@tum.de

Singleton
instance
getInstance()

Fig. 1. Specification of the Singleton pattern as it is usually found in literature.

system's architecture [TMD09]. There exist many different definitions of what constitutes an ADP. In the following, we list some of them:

"An *architectural pattern* is a named collection of *architectural design decisions* that are applicable to a recurring *design problem*, parameterized to account for different software development contexts in which that problem appears." [TMD09]

"An *architectural pattern* expresses a fundamental structural organization schema for software systems. It provides a set of predefined subsystems, specifies their responsibilities, and includes *rules and guidelines* for organizing the relationships between them." [BMR+96]

"An *architectural pattern* is a description of element and relation types together with a set of constraints on how they may be used. A pattern can be thought of as a *set of constraints* on an architecture - on the element types and their patterns of interaction - and these constraints define a set or family of architectures that satisfy them." [BCK07]

"An *architectural style* [...] defines a family of [...] systems in terms of a pattern of structural organization. More specifically, an architectural style defines a vocabulary of components and connector types, and a *set of constraints* on how they can be combined." [SG96]

Although these definitions vary in some aspects, two characteristic properties about ADPs can be identified:

1. An ADP solves a recurring *architectural design problem*.
2. An ADP consists of a collection of *architectural design constraints* which restrict the design of an architecture.

In the following, we demonstrate this observation by means of three prominent examples: the *Singleton* pattern, the *Publisher-Subscriber* pattern, and the *Blackboard* pattern.

Example 1 (The Singleton Pattern). A very basic example of an ADP, used often in object-oriented systems, is the so-called Singleton pattern [GHJV94]. It aims to address the *problem* that a system must have at most one active component of a certain type at all points in time.

If we look at a Singleton's specification, we usually find a diagram similar to the one depicted in Fig. 1. The diagram is accompanied by a description explaining that "instance" contains an instance of the singleton which can be accessed through the interface "*getInstance*()". Moreover, the description poses a *constraint* on an architecture, requiring that a new instance of type singleton is only created if no instance exists yet.

Example 2 (The Publisher-Subscriber Pattern). Another ADP often employed to design architectures is the so-called Publisher-Subscriber pattern. It aims to address the *problem* of obtaining a "flexible way of communication" between certain components of an architecture. Flexibility, in this context, means that a component can register for certain events at other components in order to be notified about the occurrence of such events.

The pattern is usually depicted with a diagram similar to the one found in Fig.2. The description usually requires the existence of two types of components: publishers and subscribers. Thereby, subscribers can subscribe for certain events and publishers are able to publish messages associated with an event. Moreover, the description usually poses a *constraint* on the connection between publisher and subscriber components which requires that, whenever a publisher component publishes a message associated with an event for which a subscriber component is currently registered, a connection between the corresponding publisher and subscriber component needs to be established.

Example 3 (The Blackboard Pattern). Another, more complex, pattern found in literature is the Blackboard pattern [TMD09, BMR+96, SG96] which is often employed for the design of systems solving logical equations.

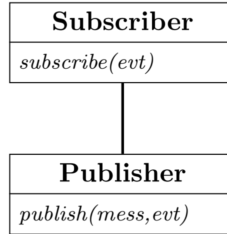


Fig. 2. Specification of the Publisher-Subscriber pattern as it is usually found in literature.

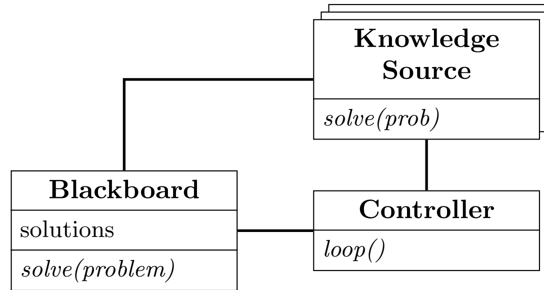


Fig. 3. Specification of the Blackboard pattern as it is usually found in literature.

The Blackboard pattern aims to address the design *problem* known as "collaborative problem solving" [1]: it is desired to design an architecture for a system which can solve a complex problem by breaking it down into simpler subproblems, which can be solved and assembled to a solution for the original problem. For example, solving a complex, logical equation (involving multiple operators), can be split into the problem of solving simpler sub-formulas and combining their solutions according to the involved logical operators.

Figure 3 shows an architecture diagram for the Blackboard pattern as it is usually found in literature. The pattern requires from an architecture the existence of the following types of components: blackboards, knowledge sources, and an optional controller component. Thereby, a blackboard keeps the overall state towards solving the original problem and knowledge sources are able to solve specific subproblems. Amongst others, the pattern requires that knowledge sources communicate exclusively through the blackboard component: they either provide solutions to currently open subproblems (given that solutions for other subproblems are available), or they communicate their ability to solve open subproblems and require a set of other subproblems to be solved first. The controller component is optional and can be employed to improve the communication between blackboard and knowledge sources.

1.1. Architectural Design Patterns

For the scope of this text, we define an ADP as follows:

Definition: *Architectural design pattern.*

An architectural design pattern consists of a specification in terms of a set of *architectural constraints*, i.e., constraints about different aspects of an architecture. Moreover, it comes with a set of *architectural guarantees* for an architecture implementing the pattern.

In this context, *architectural constraints* concern different aspects of an architecture:

- The types of *data* exchanged by the components.

[1] Problem in this context is different from architectural design problem.

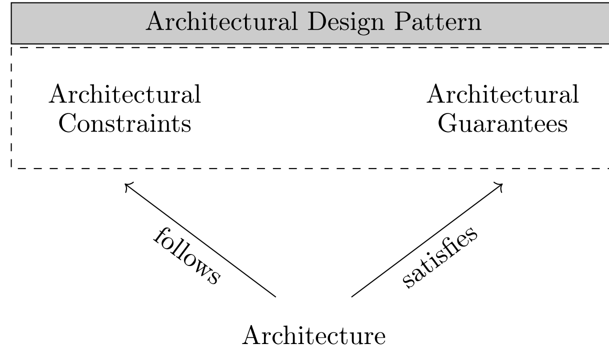


Fig. 4. Architectures and ADPs.

- The types of *components* involved in an architecture (including assumptions about their syntactic and semantic interfaces) as well as the existence of components of a certain type.
- *Activation* and *deactivation* of components of certain types.
- *Connections* between components of certain types.

The *architectural guarantees*, on the other hand, represent properties for an architecture which fulfills the constraints imposed by the pattern. Architectural guarantees usually characterize *correct* solutions for the architectural design problem addressed by an ADP. Figure 4 sketches the idea behind ADPs: An architecture which follows the constraints imposed by a pattern is assumed to satisfy the guarantees provided by the pattern. Note that we are only concerned with *functional* guarantees here and we don't consider guarantees about non-functional aspects. This is an important aspect which is further discussed in Sect. 7.3.

In the following, we demonstrate the definition by means of the three example patterns introduced above[2].

Example 4 (The Singleton Pattern). Let's first consider the Singleton pattern introduced in Ex. 1 and reformulate it in terms of our new definition. The constraints imposed by the pattern concern two aspects of an architecture: the types of components as well as the activation and deactivation of components. Thus, we may formulate two corresponding types of architectural constraints:

- A Singleton pattern requires the existence of one type of component: the singleton. However, it does not pose any constraints on the interface of a singleton component and as a consequence, it does also not constrain its behavior.
- In addition, our version of the Singleton pattern requires that a component of type singleton is always active and only one component of type singleton is active at each point in time. Note that other versions of the Singleton pattern may require that *at most* one component of type singleton is active at each point in time. Moreover, in our version of the Singleton pattern, we also require that the active component of type singleton is unique over time, i.e., the component does not change over time. Also here, we could think of different versions of the pattern, in which, for example, the singleton component is allowed to change over time.

If we try to formulate a solution for the design problem addressed by the pattern, we may end up with the following architectural guarantee (in terms of a safety property) for an architecture implementing the Singleton pattern: *A system implementing the Singleton pattern is guaranteed to have a* unique *component of type singleton which is active at each point in time.*

Example 5 (The Publisher-Subscriber Pattern). Let's now turn to the Publisher-Subscriber pattern and derive architectural constraints and architectural guarantees from the pattern's description provided in Ex. 2. In contrast to a Singleton pattern, a Publisher-Subscriber pattern constrains three aspects of an architecture: data types, component types, and connections between the ports of certain components. In the following, we provide corresponding architectural constraints:

[2] We provide only informal specifications here. Corresponding formalizations are provided in Sect. 3.

- A Publisher-Subscriber pattern requires the existence of an abstract data type to exchange subscriptions and unsubscriptions for certain events.
- In addition, a Publisher-Subscriber pattern requires two types of components: publisher and subscriber components. Although we do require the existence of certain ports for publisher as well as subscriber components, we do not require any assumptions about the behavior of these components.
- Finally, a Publisher-Subscriber pattern requires that, whenever a publisher component sends a message associated with an event for which a subscriber component is registered, the subscriber must be connected to the publisher, i.e, a channel between the corresponding ports of the publisher and subscriber component must be active in such a situation.

The following guarantee may be derived from the pattern's addressed design problem: *A subscriber receives all the messages associated with an event for which it is subscribed.* This time, the guarantee is given in terms of a liveness property for architectures implementing the pattern.

Example 6 (The Blackboard Pattern). Finally, let's derive some architectural constraints from the description of the Blackboard pattern presented in Ex. 3. Compared to the other examples, our version of the Blackboard pattern constraints every aspect of an architecture: data types, component types, component activation, and connections between components:

- First of all, a Blackboard pattern requires the existence of data types for the problems to be solved and the corresponding solutions. It even requires the existence of a *well-founded* relation between problems and corresponding subproblems.
- Moreover, a Blackboard pattern requires the existence of two types of components: blackboards and knowledge sources. Thereby, it requires that blackboard components forward the current state towards solving the original problem, i.e., they are required to forward currently open subproblems as well as solutions for already solved subproblems. Knowledge source components, on the other hand, are required to solve a problem, whenever solutions for all the required subproblems are available. Moreover, they are also required to communicate their ability to solve an open subproblem, given that solutions for other subproblems are available.
- To guarantee that a Blackboard architecture indeed solves a given problem, the pattern requires that a blackboard component is unique and always activated. Moreover, the pattern requires that for every open subproblem, a knowledge source able to solve this problem is eventually activated.
- Finally, a Blackboard pattern constraints also possible connections between blackboard and knowledge source components: whenever a knowledge source publishes a solution to an open problem, or subproblems it requires to solve an open problem, the pattern requires the knowledge source to be connected to the blackboard component.

A guarantee provided by the Blackboard pattern may be stated as follows: *An architecture is guaranteed to (collaboratively) solve a given problem, even if no knowledge source can solve the problem on its own.* Again, it is a liveness property formalizing a correct solution to the pattern's addressed design problem.

1.2. Problem: Unverified Patterns

The main problem addressed with the work presented in this paper is that patterns found in current literature are usually not verified. Figure 5 depicts this situation: it is not clear, whether the architectural constraints imposed by a pattern indeed lead to the claimed guarantee. There are two possible consequences of this problem:

- The constraints may be too weak for the guarantee.
- The constraints may be unnecessarily strong for the provided guarantee.

1.2.1. Missing Constraints

If the architectural constraints required by an ADP are too weak to ensure the claimed guarantees, the pattern may not *correctly* solve the addressed design problem. Important constraints might be missing and architectures implementing the pattern may not satisfy the pattern's guarantees.

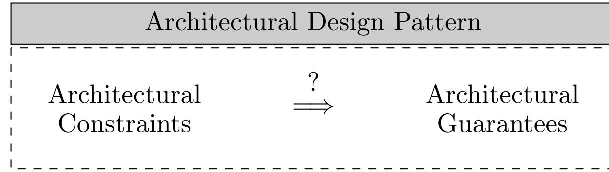


Fig. 5. Problem: unverified patterns.

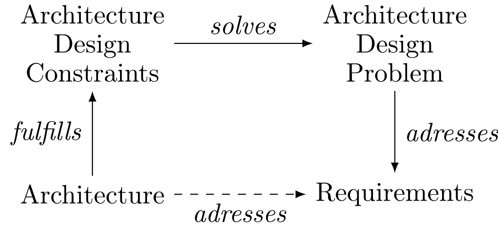


Fig. 6. The use of ADPs for the design of architectures.

To understand why missing design constraints indeed constitute a real problem, let's look at how ADPs are usually used for the design of an architecture. The situation is sketched in Fig. 6: When designing an architecture based on some requirements, ADPs are usually selected based on the problem they solve. The architecture is then designed according to the constraints imposed by the pattern. If the constraints, however, do not solve the problem, the corresponding architecture does not correctly solve the problem either, which might lead to a system which does not fulfill its requirements.

1.2.2. Unnecessary Constraints

Even if the constraints required by an ADP are strong enough to ensure a pattern's guarantee, not all constraints may be needed to ensure the pattern's guarantee. Thus, a pattern's specification may contain unnecessary constraints. While this problem is not as severe as the problem of missing constraints mentioned above, it does also have some undesired consequences: Since architectural design constraints restrict the design space for an architecture, unnecessary constraints exclude possible designs for an architecture. This becomes a problem if such an unnecessary constraint excludes an optimal design and requires an architect to select only a suboptimal architecture for the given requirements.

1.3. Approach

Over the last decade, several so-called architecture description languages (ADLs) emerged to support the formal specification and analysis of software architectures [GR91, LKA+95, All97, GMW00, DVdHT01, FLV06, HF10]. Some of those even support the specification of dynamic aspects [MK96, ADG98, vOvdLKM00]. These techniques usually specify an architecture using some type of state-machine and the specification is then analyzed using model-checking techniques.Traditional approaches to address the problems identified above tried to apply these techniques, developed for the specification and verification of architectures, to ADPs. Kim and Garlan [KG06], for example, apply the Alloy [Jac02] model checker to automatically verify architectural styles specified in ACME [Gar03]. First approaches in this area come from some early attempts to formalize design patterns using UML [MCL04, SH04, ZA05]. A similar approach comes from Wong et al. [WSWS08] which applies Alloy for the verification of architecture patterns. Zhang et al. [ZLS+12] applied model checking techniques to verify architectural styles formulated in Wright#, an extension of Wright [All97]. More recently, Marmsoler and Degenhardt [MD17] apply the NuSMV model checker [CCGR00] to verify properties of design patterns and Goethel et al. [GJS17] model patterns for self-adaptive systems using CSP [Hoa78] and use the FDR3 model checker [GRABR14] to analyze them.

Specifications of ADPs, however, have some peculiarities, which limit the applicability of the above techniques for their specification and verification: (i) Pattern specifications are usually axiomatic, focusing on a minimal set of constraints (about component behavior or architecture configurations), in order to

ensure its guarantee. For example, in a Singleton pattern, we don't care about the concrete implementation of component activation, as long as it is guaranteed that a component of type singleton is only activated if no other component of that type is already active. For a Publisher-Subscriber pattern, on the other hand, we are not interested in the concrete mechanism which implements communication between components, as long as it is guaranteed that a subscriber component is connected to a publisher, whenever latter sends out some message for which the former is currently subscribed. Or in a Blackboard pattern, we are not concerned with how a knowledge source solves a certain problem, as long as it is guaranteed that it solves it somehow. (ii) Moreover, the specification of a pattern does not necessarily contain a fixed number of components. Rather it provides upper or lower bounds and sometimes the number might be even unbounded. For example, in a Publisher-Subscriber pattern, we don't know the exact number of subscriber components. Or in a Blackboard pattern, we don't know the exact number of knowledge source components.

In this study, we propose an approach based on *axiomatic* specification techniques and *interactive theorem proving*, to address the problems identified above. Interactive theorem proving supports verification at an axiomatic level. This allows for the verification of the axiomatic specifications inherent in ADPs. The additional effort which comes with interactive theorem proving (compared to automatic verification techniques, employed in traditional approaches), is justified by the impact of verification results at pattern level: *Each result obtained at the level of ADP applies to every architecture which implements that pattern.* Thus, if we think about how many architectures implement a Singleton or a Publisher-Subscriber pattern, this should justify the additional effort induced by manual verification approaches.

1.4. Contributions

With our work, we provide the following contributions:

- We developed a language for the formal specification of architectural design constraints and architectural guarantees. To support the formal analysis of such specifications, we provide formal syntax and semantics for the language in terms of a formal model for dynamic architectures. To support a user in the development of specifications, we also implemented the language in terms of an Eclipse/EMF application.
- We developed an algorithm to systematically transfer specifications created in our language to a corresponding Isabelle/HOL theory. We showed that the algorithm preserves the semantics of the language and implemented it in Eclipse/EMF to support the automatic generation of Isabelle/HOL theories for specifications developed in our Eclipse/EMF application.
- We evaluated the approach by means of four case studies: the Singleton pattern, the Publisher-Subscriber pattern, the Blackboard pattern, and a pattern for Blockchain architectures.

> Thereby, to the best of our knowledge, this is the first attempt to apply *interactive theorem proving* for the *verification of architectural design patterns.*

With our research, we contribute to the area of *architecture verification* and the *theory of ADPs*. Since, from a practical point of view, our approach can be used to support the verification of architectures. To this end, verified patterns take the role of lemmata for the verification: they are verified once and then reused for the verification of different architectures. The fact that patterns are frequently used for the design of architectures, makes them good candidates for such lemmata.

From a theoretical point of view, the approach can help to gain more insights into the theory behind certain patterns. For, formalizing and verifying them, may reveal interesting questions about their nature. For example, when formalizing the Singleton pattern we were faced with the question whether the single instance of a component needs to be *unique* or whether it is allowed to change over time. Or, when verifying the Blockchain pattern, we discovered a fundamental assumption about the patterns environment, which is required to ensure proper functioning.

1.5. Outline

In the following, we first introduce a formal model for dynamic architectures which serves as a basis for our approach. In Sect. 3 we then present techniques to specify ADPs over this model and demonstrate them

by means of our three running examples. We then present our framework for the interactive verification of ADPs based on Isabelle/HOL. To this end, we first describe a formalization of our model in Isabelle/HOL. Then, we propose an algorithm to map a pattern specification to a corresponding Isabelle/HOL theory using the formalization of the model. Again, we demonstrate each step in terms of our three running examples. In Sect. 5 we finally integrate all our results into a methodology for the interactive specification and verification of ADPs. To this end, we also present our Eclipse/EMF-based implementation and a description of the evaluation of our approach by means of four case studies. Finally, we discuss related work and conclude the paper with a brief outlook which leads to future work.

2. A Model of Dynamic Architectures

In the following section we summarize our model for dynamic architectures [MG16b, MG16a]. The model is based on Broy's Focus theory [Bro10] and its dynamic extension [Bro14]. It assumes a set of ports and messages to be given, together with a corresponding type function which assigns a set of messages to each port. Then, it defines the notion of an *interface*, which consists of a set of input and output ports. It then introduces the central notion of *component type*, which extends an interface with a set of so-called *component parameters* (formally represented as a set of ports and associated messages) and behavior. Behavior of component types is modeled in terms of sets of so-called *behavior traces*, i.e., streams of valuations of the ports of the component's interface. Besides component types, the model introduces the concept of an *architecture trace*: streams of so-called *architecture snapshots*, which, in turn, consist of a set of active components (belonging to some type), connections between their ports, and a valuation of the ports of active components. An *architecture specification* is then defined as a set of architecture traces which does not restrict the behavior of components. Finally, the notion of *behavior projection* is introduced to extract the behavior of a certain component out of a given architecture trace. Behavior projection is then used to define composition of component types under a given architecture specification: the result of composing a set of component types with an architecture specification is defined to consist of all the architecture traces from the architecture specification, for which the projection to any component leads to a behavior trace allowed by the component's type.

In the following, we first introduce the basic concepts of messages, ports, and interfaces. Then, we describe two key concepts of our model: component types and architecture specifications. Finally, we introduce the notion of behavior projection to define composition of component types with architecture specifications. We conclude with a summary of the introduced concepts and their interrelationships.

2.1. Messages and Ports

In our model, components communicate with 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.

2.2. Port Valuations

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:

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

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, $\ldots$,

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

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.

2.3. Interfaces

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 (I, O), consisting of *disjoint* sets of *input ports* $I \subseteq \mathcal{P}$ and *output ports* $O \subseteq \mathcal{P}$. The set of all interfaces is denoted by $IF_{\mathcal{P}}$. For an interface $\mathit{if} = (I, O)$, we denote by

- $\mathsf{in}(\mathit{if}) \stackrel{\text{def}}{=} I$ the set of input ports,
- $\mathsf{out}(\mathit{if}) \stackrel{\text{def}}{=} O$ the set of output ports, and
- $\mathsf{port}(\mathit{if}) \stackrel{\text{def}}{=} I \cup O$ the set of all interface ports.

2.4. Streams

In the following, we shall make use of finite as well as infinite *streams* [BS01]. 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 $s(n-1)$ 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 $s\hat{\ }s'$ we denote the concatenation of stream s with stream s'. For the case s is infinite, $s\hat{\ }s' = s$.
- 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 $s\downarrow_0 \stackrel{\text{def}}{=} \langle\rangle$.
- With $s' \sqsubseteq s$, we denote that s' is a prefix of s.

2.5. Component Types

A key concept of our model is the concept of a component type, i.e., a parameterized interface with associated behavior. The behavior is given in terms of so-called behavior traces, i.e., streams of valuations of the corresponding interface ports.

Definition 3 (Component type). A *component type* is a 4-tuple $(\mathit{if}, CP, \mu, bhv)$, consisting of

- an interface $\mathit{if} \in IF_{\mathcal{P}}$,
- so-called *component parameters* $CP \subseteq \mathcal{P}$ which are required to be disjoint from the interface's input and output ports,
- a valuation of the component parameters $\mu \in \overline{CP}$,
- and a set of so-called *behavior traces*, $bhv \subseteq (\overline{\mathsf{port}(\mathit{if})})^\infty$.

The set of all possible component types over a set of interfaces $\mathcal{I} \subseteq IF_{\mathcal{P}}$ is denoted $CT_{\mathcal{I}}$. We shall use the same notation as introduced for interfaces in Def. 2, to denote input, output, and all interface ports for component types. Moreover, for a component type $(\mathit{if}, CP, \mu, bhv)$, we denote by

- $\mathsf{par}(ct) \stackrel{\text{def}}{=} CP$ its component parameters,

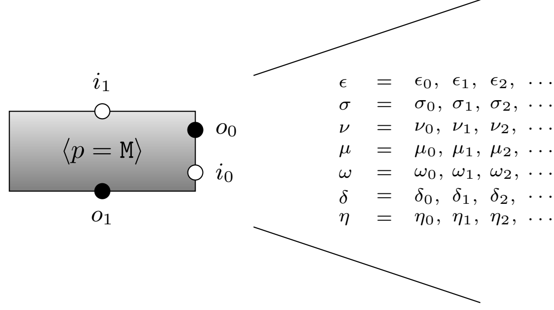


Fig. 7. Conceptual representation of a component type with component parameter p, valuated by $\texttt{M}$, and behavior $bhv = \{\epsilon, \sigma, \nu, \mu, \omega, \delta, \eta\}$.

- $\mathsf{val}(ct) \stackrel{\text{def}}{=} \mu$ the valuation of component parameters, and
- $\mathsf{bhv}(ct) \stackrel{\text{def}}{=} bhv$ the behavior of a component type $ct = (\mathit{if}, CP, \mu, bhv)$.

Example 7 (Component type). Assuming $\mathcal{P}$ contains ports p, i_0, i_1, o_0, o_1. Figure 7 shows a conceptual representation of a component type $(\mathit{if}, CP, \mu, bhv)$, consisting of:

- Interface $\mathit{if} = (I, O)$, with
  - input ports $I = \{i_0, i_1\}$, and
  - output ports $O = \{o_0, o_1\}$.
- Component parameters $CP = \{p\}$.
- Parameter valuation $\mu(p) = \texttt{M}$.
- Behavior $bhv = \{\epsilon, \sigma, \nu, \mu, \omega, \delta, \eta\}$, consisting of behavior traces
  - $\epsilon = \epsilon_0,\ \epsilon_1,\ \epsilon_2,\ \ldots$
  - $\sigma = \sigma_0,\ \sigma_1,\ \sigma_2,\ \ldots$
  - $\ldots$

Component types specify allowed behavior for components of a certain type. Thereby, they introduce a notion of parametrization as a means to support the specification of groups of related components of a certain type. Since they are mainly a concept to support the specification of component types, component parameters are discussed in more detail in the next chapter.

2.6. Architecture Specifications

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 specification to address these aspects.

2.6.1. 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, valuation of component parameters, 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:

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

Moreover, we may lift the typing function (introduced for ports in Sect. 2.1), to corresponding component ports:

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

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*:

$$\overline{CP} \stackrel{\text{def}}{=} \Big\{\mu \in \big(CP \to \wp(\mathcal{M})\big) \mid \forall cp \in CP\colon \mu(cp) \subseteq \mathcal{T}(cp)\Big\} \ .$$

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.

2.6.2. Architecture Snapshots

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

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

- a set of active components $C' \subseteq \mathcal{C}$,
- a connection $N\colon \mathsf{in}(C') \to \wp(\mathsf{out}(C'))$, such that types of connected ports are compatible:

$$\forall ci \in \mathsf{in}(C')\colon \bigcup_{co \in N(ci)} \mathcal{T}(co) \subseteq \mathcal{T}(ci) \ , \text{ and} \tag{2}$$

- 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:

$$\forall ci \in \mathsf{in}(C')\colon N(ci) \neq \emptyset \implies \mu(ci) = \bigcup_{co \in N(ci)} \mu(co) \ . \tag{3}$$

Note that Eq. (2) guarantees that Eq. (3) 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

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

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

$$\mathsf{cmp}^c_{as} \in \overline{\mathsf{port}(c)} \stackrel{\text{def}}{=} \Big(\lambda p \in \mathsf{port}(c)\colon \mu\big((c,p)\big)\Big) \tag{4}$$

the valuation of the component's ports.

Note that cmp^c_{as} is well-defined iff $|c|_{as}$.

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).

Moreover, note that by Eq. (3), 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 8 (Architecture snapshot). Figure 8 shows a conceptual representation of an architecture snapshot (C', N, μ), consisting of:

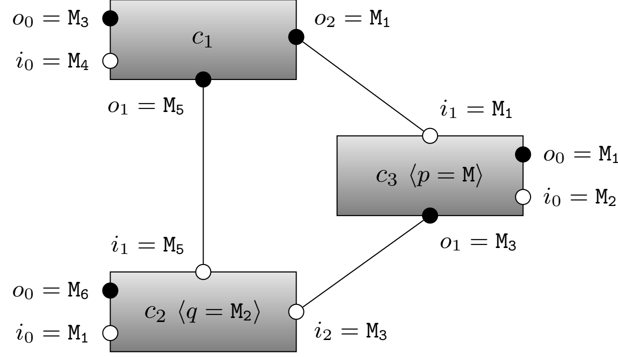


Fig. 8. Architecture snapshot consisting of three components c_1, c_2, and c_3; a connection between ports (c_2, i_1) and (c_1, o_1), (c_2, i_2) and (c_3, o_1), and (c_3, i_1) and (c_1, o_2); and valuations of the component parameters and ports.

- active components $C' = \{c_1, c_2, c_3\}$ with corresponding component types (c_3, for example, could be of the type as described in Ex. 7);
- 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 $[(c_1, o_0), (c_2, i_1), (c_3, o_1), \cdots \mapsto \mathtt{M}_3, \mathtt{M}_5, \mathtt{M}_3, \cdots]$.

2.6.3. 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 an infinite stream $t \in (AS_{\mathcal{T}}^{\mathcal{C}})^{\infty}$. For an architecture trace $t \in (AS_{\mathcal{T}}^{\mathcal{C}})^{\infty}$ and a component $c \in \mathcal{C}$, we denote with

- *last*(c, t), the *greatest* $i \in \mathbb{N}$, such that $|c|_{t(i)}$,
- $c \overset{n}{\Leftarrow} t$, 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 $n' = n \vee \left(n' < n \wedge \nexists n' \leq k < n \colon |c|_{t(k)}\right)$,
- $c \overset{n}{\leftarrow} t$, 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 $c \overset{n}{\Leftarrow} t$ is always well-defined, while $c \overset{n}{\leftarrow} t$ and $c \overset{n}{\rightarrow} t$ are only well-defined iff there exists at least one activation of component c in the past (a point in time strictly less than n) or in the future (a point in time greater or equal to n), respectively. *last*(c, t), on the other hand, is well-defined iff i) component c is activated at least once in t: $\exists i \in \mathbb{N} \colon |c|_{t(i)}$ and ii) component c is not activated infinitely often, i.e., $\exists n \in \mathbb{N} \colon \forall n' \geq n \colon \neg|c|_{t(n')}$.

Example 9 (Architecture trace). Figure 9 shows an architecture trace $t \in (AS_{\mathcal{T}}^{\mathcal{C}})^{\infty}$ 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. 8.

Figure 10 lists some properties derived for the operators introduced for architecture traces in Def. 5.

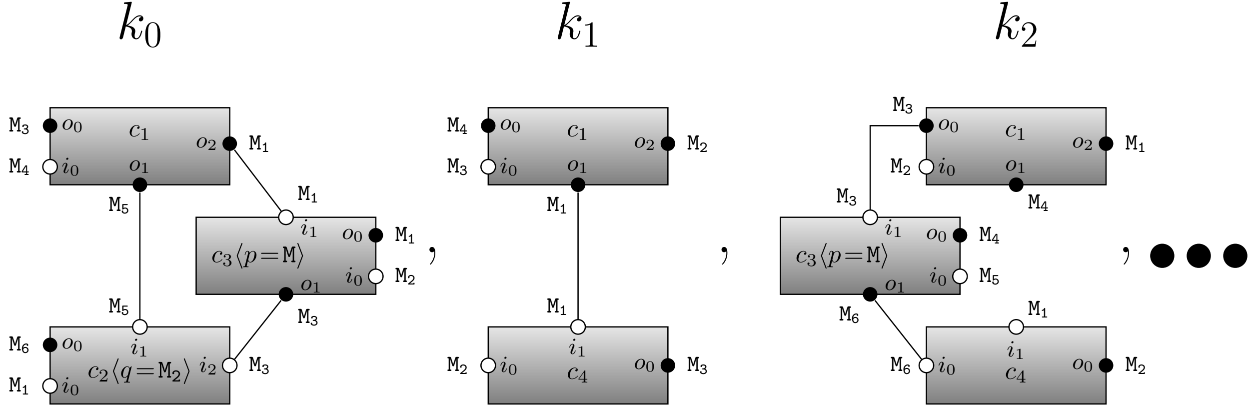


Fig. 9. The first three architecture snapshots of an architecture trace.

Properties of component activation

$$
\begin{aligned}
|c|_{t(last(c,t))} &\;\left[\text{if } \exists i\colon |c|_{t(i)} \text{ and } \exists n\colon \forall n' > n\colon \neg|c|_{t(n')}\right]\\
\nexists n > last(c,t)\colon |c|_{t(n)} &\;\left[\text{if } \exists n'\colon \forall n'' > n'\colon \neg|c|_{t(n'')}\right]\\
c \overset{0}{\Leftarrow} t = 0 &\\
|c|_{t(n-1)} \implies c \overset{n}{\Leftarrow} t = n &\;\left[\text{if } n \geq 1\right]\\
\forall c \overset{n}{\Leftarrow} t \leq n' < n\colon \neg|c|_{t(n')} &\\
c \overset{n}{\Leftarrow} t \leq n &\\
c \overset{c \overset{n}{\Leftarrow} t}{\rightarrow} t = c \overset{n}{\leftarrow} t &\;\left[\text{if } \exists i < n\colon |c|_{t(i)}\right]\\
c \overset{n}{\rightarrow} t > c \overset{n}{\leftarrow} t &\;\left[\text{if } \exists i \geq n\colon |c|_{t(i)} \text{ and } \exists i < n\colon |c|_{t(i)}\right]\\
c \overset{n}{\rightarrow} t \geq n &\;\left[\text{if } \exists i \geq n\colon |c|_{t(i)}\right]\\
|c|_{t(c \overset{n}{\rightarrow} t)} &\;\left[\text{if } \exists i \geq n\colon |c|_{t(i)}\right]\\
\nexists n \leq k < c \overset{n}{\rightarrow} t\colon |c|_{t(k)} &\;\left[\text{if } \exists i \geq n\colon |c|_{t(i)}\right]\\
|c|_{t(n)} \implies c \overset{n}{\rightarrow} t = n &\\
c \overset{n}{\rightarrow} t \geq c \overset{n}{\Leftarrow} t &\;\left[\text{if } \exists i \geq n\colon |c|_{t(i)}\right]
\end{aligned}
$$

Fig. 10. Properties of component activation.

2.6.4. Architecture Specifications

Finally, we can define our notion of architecture specification as a set of architecture traces with certain properties.

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

$$
\begin{aligned}
&\forall t \in \mathcal{A},\ n \in \mathbb{N},\ \mu \in \overline{\mathsf{out}(CMP_{t(n)})}\ \exists t' \in \mathcal{A}\colon t'{\downarrow_n} = t{\downarrow_n}\\
&\qquad \wedge CMP_{t(n)} = CMP_{t'(n)} \ \wedge CN_{t(n)} = CN_{t'(n)} \ \wedge val_{t'(n)}|_{\mathsf{out}(CMP_{t(n)})} = \mu \qquad (5)
\end{aligned}
$$

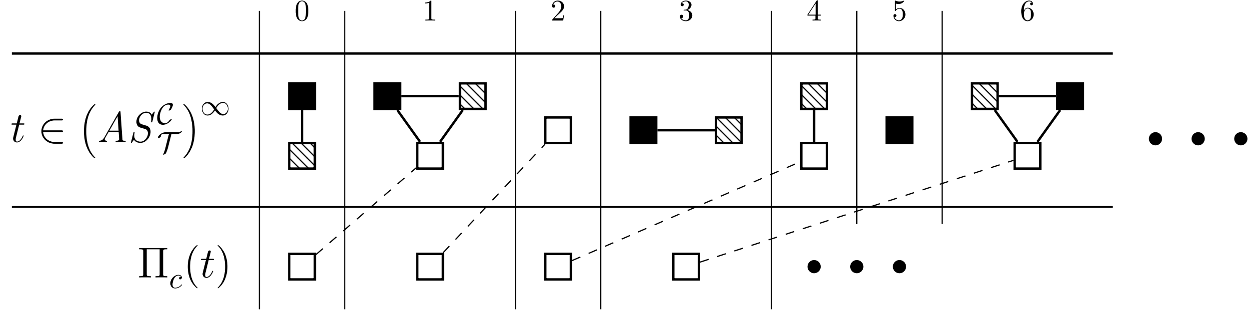


Fig. 11. Conceptual representation of behavior projection.

2.7. Behavior Projection and Composition

Note that an architecture specification does not restrict the behavior of components. A component's behavior, on the other hand, is restricted in the specification of component types. We conclude the section by introducing the notion of composition as a means to combine a specification of component types with a corresponding architecture specification. First, however, we introduce the concept of behavior projection, which is essential for the definition of composition.

2.7.1. Behavior projection

An important concept for our model is the notion of behavior projection. It is used to extract the behavior of a certain component out of a given architecture trace (Figure 11).

In the following, we provide a *co-recursive* definition for behavior projection. This allows us to easily specify the operator also for infinite input traces by following a certain pattern in its specification. Then, we can use co-induction to reason about behavior projection[3].

Definition 7 (Behavior projection). Given an architecture trace $t \in (AS^{\mathcal{C}}_{\mathcal{T}})^{\omega}$. 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:

$$\Pi_c(\langle\rangle) = \langle\rangle \quad (6)$$

$$|c|_{as} \implies \Pi_c(as \mathbin{\&} t) = \mathsf{cmp}^c_{as} \mathbin{\&} \Pi_c(t) \quad (7)$$

$$\neg|c|_{as} \implies \Pi_c(as \mathbin{\&} t) = \Pi_c(t) \quad (8)$$

$$\left(\forall as \in rg(t)\colon \neg|c|_{as}\right) \implies \Pi_c(t) = \langle\rangle \quad (9)$$

Note that the structure of the equations provided in Def. 7 ensures productivity [JR97] and hence they resemble a valid *co-recursive* definition. Thus, projection is indeed well-defined by those equations.

Example 10 (Behavior projection). Applying behavior projection of component c_3 to the architecture trace described in Ex. 9 results in a behavior trace starting as follows:

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

Figure 12 lists some characteristic properties of behavior projection.

2.7.2. Composition

Behavior projection can now be used to define composition of components according to an architecture specification.

[3] Alternatively we could have used traditional recursion, show that behavior projection is continuous, and use fixpoint induction [GH05] to proof properties about it. The reason to choose co-recursion here is that it simplifies subsequent formalization in Isabelle/HOL.

Properties of behavior projection

$$
\begin{aligned}
\#\Pi_c(t) \quad &\leq \quad \#t \\
\Pi_c(t) \quad &= \quad \Pi_c(t{\downarrow}_n) \text{ [if } \forall n \leq n' \leq \#t \colon \neg|c|_{t(n')}] \\
\mathit{finite}(\Pi_c(t)) \quad &\Longleftrightarrow \quad \exists n\ \forall n' > n \colon \neg|c|_{t(n')} \\
t \sqsubseteq t' \quad &\Longrightarrow \quad \Pi_c(t) \sqsubseteq \Pi_c(t') \\
\Pi_c(t ^\frown t') \quad &= \quad \Pi_c(t) ^\frown \Pi_c(t') \text{ [if } \mathit{finite}(t)] \\
\Pi_c(t{\downarrow}_{n+1}) \quad &= \quad \Pi_c(t{\downarrow}_n) \text{ [if } n < \#t \text{ and } \neg|c|_{t(n)}] \\
\Pi_c(t{\downarrow}_{i+1}) \quad &= \quad \Pi_c(t{\downarrow}_i) ^\frown \mathsf{cmp}^c_{t(i)}\ \&\ \langle\rangle \text{ [if } i < \#t \text{ and } |c|_{t(i)}]
\end{aligned}
$$

Fig. 12. Properties of behavior projection.

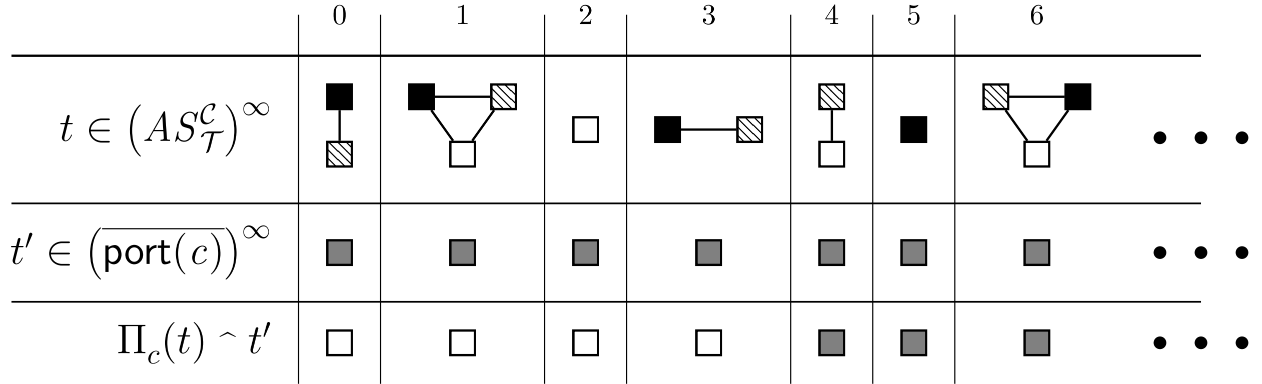


Fig. 13. Continuations for unfair architecture traces.

Definition 8 (Composition). *Composition* of a family of components $(\mathcal{C}_{ct})_{ct \in \mathcal{CT}}$ with an architecture specification $\mathcal{A} \subseteq (AS^{\mathcal{C}}_{\mathcal{T}})^{\infty}$, is defined as follows:

$$
\bigotimes_{\mathcal{A}}(\mathcal{C}) \quad \overset{\text{def}}{=} \quad \left\{ t \in \mathcal{A} \mid \forall ct \in CT,\ c \in C_{ct}\ \exists t' \subseteq (\overline{\mathsf{port}(ct)})^{\infty} \colon \Pi_c(t) ^\frown t' \in \mathsf{bhv}(ct) \right\} \tag{10}
$$

Note that the projection to an *unfair* architecture trace t, i.e., a trace in which a component is activated only finitely many times, the projection to this component results in only a finite behavior trace. Thus, we actually search for a valid *continuation* t', such that the concatenation for the projection $\Pi_c(t)$ with t' is a valid behavior of c. The situation is depicted in Fig. 13: The projection to component c (represented by the empty rectangle) in architecture trace t, is combined with a possible continuation t' to obtain an *infinite* behavior trace $\Pi_c(t) ^\frown t'$ (shown at the bottom of Fig. 13).

2.8. Summary

Figure 14 summarizes the main concepts of our model and their interrelationships: Messages and ports (typed by sets of messages) form the basic concepts of the model. A key concept of the model is the notion of *component type* which consists of an interface (sets of input and output ports as well as component parameters) and a behavior in terms of behavior traces (streams of port valuations, i.e., valuations of ports with messages). Another important concept is the concept of *architecture specification*: a special set of architecture traces (streams of architecture snapshots, i.e., states of an architecture during execution). Finally, the model provides an operator to combine a given architecture specification with a set of component types and corresponding components. Therefore, the operator uses the concept of *behavior projection* which extracts the behavior of a certain component out of a given architecture trace.

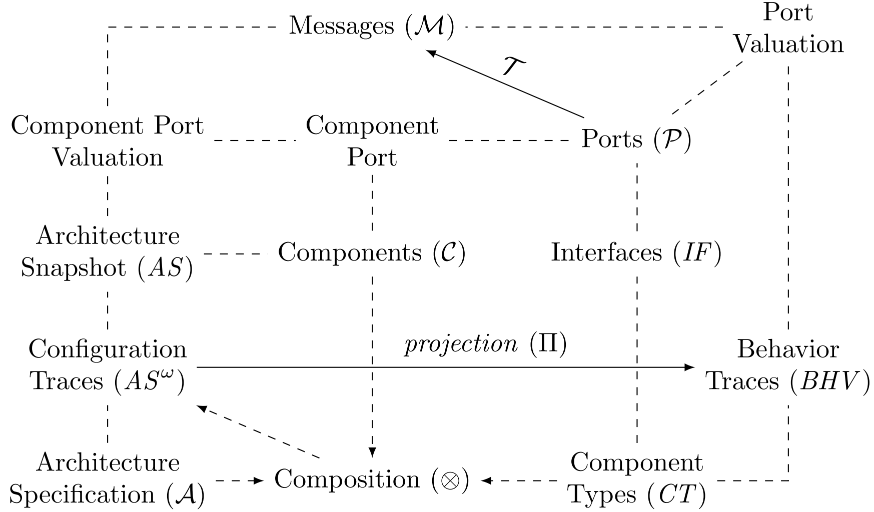


Fig. 14. Concept map summarizing major concepts and their interrelationships.

3. Specifying Architectural Design Patterns

In the previous section, we described a model for dynamic architectures, based on the concept of *component types* and *architecture specifications*. In this section, we introduce techniques to specify ADPs over this model. The techniques are summarized in Fig. 15: First, *data types* are specified for the messages exchanged by the components of an architecture using traditional, *algebraic specification techniques* [Bro96, Wir90]. Then, *component types* are specified on top of these datatypes. To this end, corresponding interfaces are specified for each type of component using a graphical notation called *architecture diagrams*. Then, component behavior is specified over these interfaces using so-called *behavior trace assertions*, i.e., linear temporal logic formulæ with ports as free variables. Finally, an *architecture specification* is given in terms of so-called *architecture trace assertions*: linear temporal logic formulæ with component variables and architecture predicates. In the following, we detail on each of the techniques and demonstrate them by means of our three running examples: the Singleton, the Publisher-Subscriber, and the Blackboard pattern.

3.1. Specifying Data Types

As a first step, a set of data types is specified for a pattern. Data types are specified in terms of axioms over a signature and corresponding variables. They can be specified using traditional, algebraic specification techniques [Bro96, Wir90]. Fig. 16 depicts a schematic example of such an algebraic specification: Each specification has a name and may be parameterized by several *sorts*. Moreover, other data type specifications can by imported by means of their name. Function/predicate *symbols* are introduced with the corresponding sorts at the beginning of the specification. Some of the symbols might be declared as *generator clauses*, requiring that every element of the corresponding datatype can be "reached" by a term formulated with these symbols. Finally, a list of variables for the different sorts is defined and a set of *axioms* is formulated over the symbols and the variables, to specify a datatype's characteristic properties.

In the following, we demonstrate datatype specifications using our three running examples. The specification of the Singleton pattern does not require any special data types. Data types are required, however, for the specification of the Publisher-Subscriber pattern as well as the Blackboard pattern.

Example 11 (Datatype specification for the Publisher-Subscriber pattern). In a Publisher-Subscriber pattern, we usually have two types of messages: *subscriptions* and *unsubscriptions*. Figure 17 depicts the corresponding data type specification. Subscriptions are modeled as *parametric* data types over two type parameters: a type `id` for component identifiers and some type `evt` denoting events to subscribe

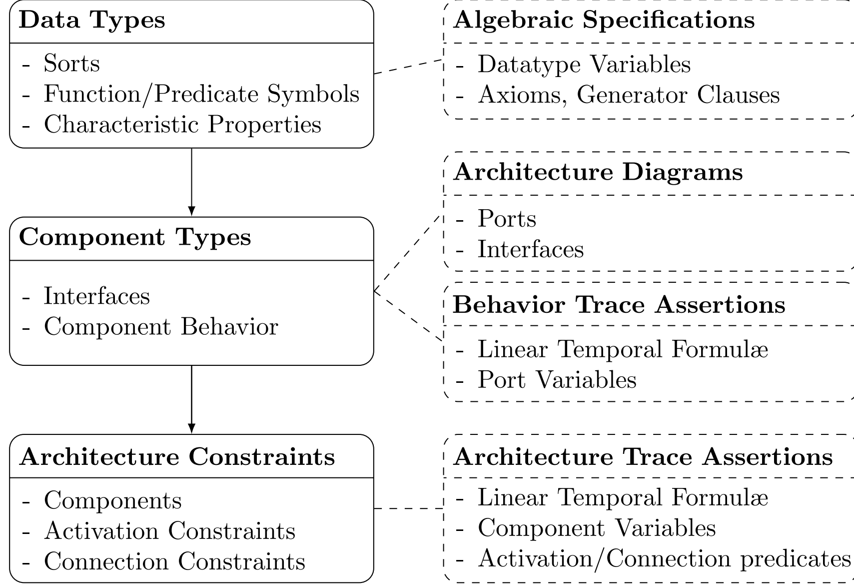


Fig. 15. Specifying architectural design patterns.

DTSpec Name(`param`)	**imports** OtherDatatype
symbol1 :	`Sort1`
symbol2 :	`Sort1` $\rightarrow$ `Sort2`
⋮	
generated by *symbol1*, *symbol2*	
⋮	
flex *var1*, *var2* :	`Sort1`
var3 :	`Sort2`
⋮	
assertion1(*symbol1*, *var1*, *var2*, *var4*)	
assertion2(*symbol1*, *symbol2*, *var1*, *var4*)	
⋮	

Fig. 16. Schematic algebraic specification for data types.

for. The data type is freely generated by the constructor terms "sub `id evt`" and "unsub `id evt`", meaning that every element of the type has the form "sub `id evt`" or "unsub `id evt`".

Example 12 (Datatype specification for the Blackboard pattern). Blackboard architectures usually work with *problems* and *solutions* for them. Figure 18 provides a specification of the corresponding data types. We denote by `PROB` the set of all problems and by `SOL` the set of all solutions. Complex problems consist of *subproblems*, which can be complex themselves. To solve a problem, its subproblems have

DTSpec subscription(`id`, `evt`)
generated by sub `id` $\wp$(`evt`), unsub `id` $\wp$(`evt`)

Fig. 17. Data type specification for the Publisher-Subscriber pattern.

DTSpec ProbSol	**imports** SET
$\prec$:	$\texttt{PROB} \times \texttt{PROB}$
solve :	$\texttt{PROB} \rightarrow \texttt{SOL}$
well-founded($\prec$)	(11)

Fig. 18. Data type specification for the Blackboard pattern.

PSpec Port Specification	**imports** Datatype
port1 :	`Sort1`
port2 :	`Sort2`
⋮	

Fig. 19. Exemplary port specification.

to be solved first. Therefore, we assume the existence of a *subproblem relation* $\prec \subseteq \texttt{PROB} \times \texttt{PROB}$. For complex problems, the *details* of the relation may not be known in advance. Indeed, one of the benefits of a Blackboard architecture is that a problem can be solved even without knowing the exact nature of this relation in advance. However, the subproblem relation has to be well-founded[4] (Eq. (11)) for a problem to be solvable. In particular, we do not allow for cycles in the transitive closure of $\prec$. While there may be different approaches to solve a problem (i.e., several ways to split a problem into subproblems), we assume that the final solution to a problem is always unique. Thus, we assume the existence of a function *solve*: $\texttt{PROB} \rightarrow \texttt{SOL}$, which assigns the *correct* solution to each problem. Note, however, that it is not known in advance *how* to compute this function and it is indeed one of the reasons for using this pattern to compute this function.

3.2. Specifying Component Types

On top of the specified data types, a set of component types (as described in Def. 3) is specified for the pattern. Component types are specified in two steps: First, an interface is specified for them using a graphical notation called architecture diagrams. Then, constraints about the behavior of components of a certain type are specified in terms of behavior trace assertions.

3.2.1. Specifying Interfaces

On top of the specified data types, a set of interfaces for the component types (as introduced in Def. 2) are specified. The specification of interfaces proceeds in two steps: First, a set of port types is specified as a means to exchange messages of a certain type. Then, interfaces are specified over these port types.

Port specifications. Port types are specified by means of templates, which declare a set of port identifiers and associate them with sorts from the corresponding datatype specification. Figure 19 shows such a template which specifies two types of ports: *port1* of type `Sort1` and *port2* of type `Sort2`, respectively.

Again, we demonstrate port specifications by means of our running examples, and again, the specification of the Singleton pattern does not require any ports, at all. However, Publisher-Subscriber architectures and also Blackboard architectures require ports to be specified.

Example 13 (Port specification for the Publisher-Subscriber pattern). Two port types are specified for the Publisher-Subscriber pattern by the specification given in Fig. 20: a type *sb* which allows to exchange subscriptions for specific events and type *nt*, which allows to exchange messages associated with a certain event. To this end, it uses a type parameter `msg` and imports the data type specification for subscriptions described in Ex. 11.

[4] A *well-founded* relation is a partial order which has no infinite decreasing chains.

PSpec PSPorts($\mathtt{msg}$)	**imports** subscription($\mathtt{id}, \mathtt{evt}$)
sb:	subscription($\mathtt{id}, \mathtt{evt}$)
nt:	$\mathtt{evt} \times \mathtt{msg}$

Fig. 20. Port specification for the Publisher-Subscriber pattern.

PSpec BBPorts	**imports** ProbSol
rp:	$\mathtt{PROB} \times \wp(\mathtt{PROB})$
ns, *cs*:	$\mathtt{PROB} \times \mathtt{SOL}$
op, *prob*:	$\wp(\mathtt{PROB})$

Fig. 21. Port specification for the Blackboard pattern.

Example 14 (Port specification for the Blackboard pattern). For the specification of the Blackboard pattern we require 4 different types of ports, specified in Fig. 21:

- *rp* is used to exchange a problem which a knowledge source is able to solve, together with a set of subproblems the knowledge source requires to be solved first.
- *ns* is used to exchange an open problem, solved by a knowledge source, together with the corresponding solution.
- *op* is used to exchange problems which still need to be solved.
- *cs* is used to exchange existing solutions for already solved problems.

Since a knowledge source can only solve certain problems, they are parameterized by a corresponding component parameter later on. Formally, however, a parameter is just another port, which needs to be specified accordingly. Thus, we specify an additional port *prob* in Fig. 21, which is used later on to parametrize knowledge sources according to the problems they can solve.

Interface specifications. Interfaces consist of a set of input and output ports. Moreover, interface specifications may also specify a set of *component parameters*. Formally, component parameters are represented as ports; however, they have a special meaning in the following sense:

- The valuation of component parameters is bound to a concrete component (as required by Def. 3), i.e., the valuation does not change over time, compared to valuations of input and output ports.
- For each possible valuation of the component parameters, at least one component is guaranteed to exist (as required in Sect. 2.6.1). This is not the case for input and output port valuations.

Interfaces are specified over a given port specification, and they are best expressed graphically using so-called *architecture diagrams*. Thereby, an interface is represented by a rectangle and consists of two parts: i) A name followed by a list of component parameters (enclosed between '⟨' and '⟩'. ii) A set of input and output ports which are represented by empty and filled circles, respectively. Figure 22 shows a conceptual representation of an architecture diagram *Name*, which is based on a port specification "PortSpecification". It specifies an interface *If1*, which consists of one input port *i*, one output port *o*, and a component parameter *par*.

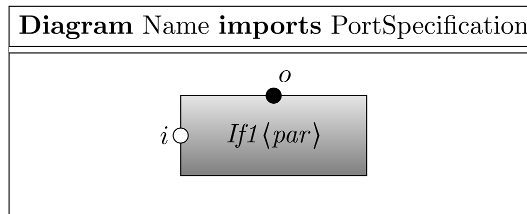


Fig. 22. Basic architecture diagram.

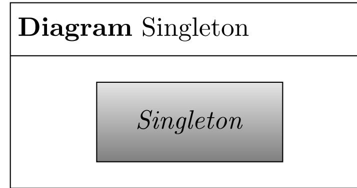


Fig. 23. Architecture diagram for the Singleton pattern.

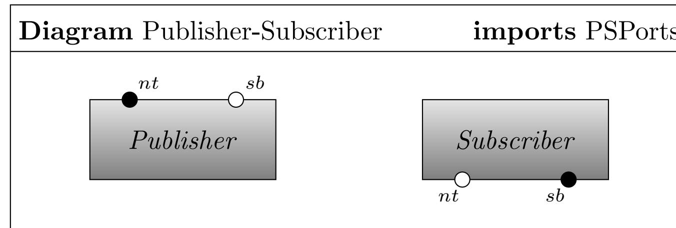


Fig. 24. Architecture diagram for the Publisher-Subscriber pattern.

Besides specifying interfaces for component types, an architecture diagram can also be used to introduce common activation and connection constraints. For now, however, you can consider them as a simple way to introduce interfaces. Later on, in Sect. 3.4, we introduce additional notations for architecture diagrams to make pattern specifications more concise.

In the following, we provide basic interface specifications for all of our running examples.

Example 15 (Interface specification for the Singleton pattern). The architecture diagram depicted in Fig. 23 specifies interfaces for the Singleton pattern: It consists of a single interface *Singleton* which does not require the existence of any special ports.

Example 16 (Interface specification for the Publisher-Subscriber pattern). The architecture diagram depicted in Fig. 24 shows the specification of the interfaces of the two types of components involved in a Publisher-Subscriber pattern: An interface *Publisher* is defined with an input port *sb* to receive subscriptions and an output port *nt* to send out notifications. Moreover, an interface *Subsciber* is defined with an input port *nt* receiving notifications and an output port *sb* to send out subscriptions. Note also that the diagram imports the specification of ports discussed in Ex. 13.

Example 17 (Interface specification for the Blackboard pattern). The specification of interfaces for the Blackboard pattern is provided by the architecture diagram depicted in Fig. 25. Again, the diagram imports the corresponding port specification from Ex. 14. Moreover, it specifies interfaces for blackboards and knowledge sources.

The blackboard interface is denoted *BB* and consists of two input ports *rp* and *ns* to receive subproblems for which solutions are required and new solutions to currently open problems, respectively. Moreover, it specifies two output ports *op* and *cs* to communicate currently open problems and solutions for all currently solved problems, respectively.

The interface for knowledge sources is denoted *KS* and its specification mirrors the specification of the blackboard interface: A knowledge source is required to have two input ports *op* and *cs* to receive currently open problems and solutions for all currently solved problems, and two output ports *rp* and *ns* to communicate required subproblems and new solutions. Note that each knowledge source can only solve certain problems, which is why a knowledge source is parameterized by a set of problems *prob* it is able to solve.

3.2.2. Specifying Behavioral Constraints

We conclude the specification of component types by assigning constraints about component behavior to each interface. These constraints are expressed in terms of so-called *behavior trace assertions*. In the following, we

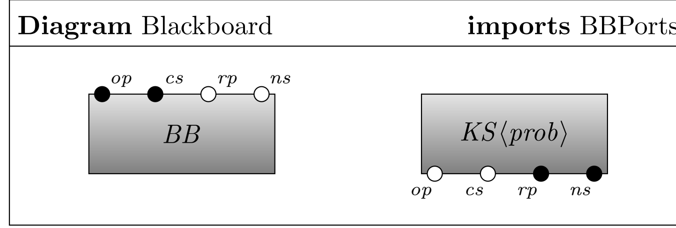


Fig. 25. Architecture diagram for the Blackboard pattern.

BSpec Name	**for** iface **of** ispec
flex aDt1 :	`Sort1`
rig aDt2 :	`Sort1`
⋮	
assertion1 (iface, aDt1, aDt2)	
assertion2 (iface, aDt1, aDt2)	
⋮	

Fig. 26. Schematic component type specification.

introduce behavior trace assertions informally and demonstrate them by means of our Blackboard example. However, in App. A, we provide also a formal definition of the syntax and semantics of behavior trace assertions.

Behavior trace assertions. Behavior trace assertions are a means to specify a component's behavior in terms of behavior traces introduced in Def. 3. They are formulated by means of first-order linear temporal logic formulæ [MP92] over *datatype variables* and *behavior assertions*.

Behavior trace assertions may be specified over variables for messages. Thereby, variables are typed by the sorts of the pattern's datatype specification and we distinguish two types of variables: *rigid* and *flexible* datatype variables. While rigid variables are only interpreted once, flexible variables are newly interpreted at each point in time.

Roughly speaking, behavior assertions are predicates specified over a set of *datatype variables* and a set of ports, with terms consisting of:

- Datatype variables as well as the ports and parameters of a component type's interface.
- Function and predicate symbols of the corresponding data type specification.

They specify the state of a component (in terms of valuations of input and output ports) during execution.

Component type specifications. Component types are specified using templates as shown in Fig. 26. The specification has a name and is associated with an interface of a corresponding interface specification. Then, a set of flexible and rigid variables for different sorts are declared. A component type specification concludes with a list of behavior trace assertions, formulated over the ports of the corresponding interface and the introduced data type variables.

In the following, we demonstrate component type specifications in terms of our running examples. However, since the Singleton pattern and the Publisher-Subscriber pattern do not pose any constraints on the behavior of components, we only provide behavior specifications for the two types of components involved in a Blackboard pattern.

Example 18 (Behavioral constraints for blackboard components). A blackboard provides the *current state* towards solving the original problem and forwards problems and solutions from knowledge sources. Fig. 27 provides a specification of the blackboard's behavior in terms of three behavior trace assertions:

- If a solution s' to a subproblem p' is received on its input port ns, then it is eventually provided at its output port cs (Eq. 12).

BSpec Blackboard	**for** BB **of** Blackboard
flex p:	`PROB`
P:	`PROB SET`
rig p':	`PROB`
s':	`SOL`

$$\Box\Big((p',s') \in ns \longrightarrow \Diamond\big((p',s') \in cs\big)\Big) \tag{12}$$

$$\Box\Big((p,P) \in rp \longrightarrow \big(\forall p' \in P\colon \ (\Diamond(p' \in op))\big)\Big) \tag{13}$$

$$\Box\Big(p' \in op \longrightarrow \big(p' \in op \ \mathcal{W} \ (p', solve(p')) \in ns\big)\Big) \tag{14}$$

Fig. 27. Specification of behavior for blackboard components.

BSpec Knowledge Source	**for** $KS\langle prob\rangle$ **of** Blackboard
flex $p, \ q$:	`PROB`
P:	$\wp$(`PROB`)
rig $p', \ q'$:	`PROB`

$$\Box\Big(\forall (p',P) \in rp\colon \ \big((\forall q' \in P\colon \ \Diamond(q', solve(q')) \in cs) \longrightarrow \Diamond(p', solve(p')) \in ns\big)\Big) \tag{15}$$

$$\Box\Big(\forall (p,P) \in rp\colon \ \forall q \in P\colon \ q \prec p\Big) \tag{16}$$

$$\Box\Big(prob \in op \longrightarrow \Diamond(\exists P\colon \ (prob, P) \in rp)\Big) \tag{17}$$

Fig. 28. Specification of behavior for knowledge source components.

- If it gets notified that solutions for subproblems P are required to solve a certain problem p on its input port rp, these problems are eventually provided at its output port op (Eq. (13)).
- A problem p' is provided at its output port op, as long as it is not solved (Eq. (14)).

Note that the last assertion (Eq. (14)) is formulated using a *weak* until operator which is defined as follows: $\gamma' \ \mathcal{W} \ \gamma \ \stackrel{\text{def}}{=} \ \Box(\gamma') \vee (\gamma' \ \mathcal{U} \ \gamma)$.

Example 19 (Behavioral constraints for knowledge source components). A knowledge source receives open problems and solutions for already solved problems. It might contribute to the solution of the original problem by solving currently open subproblems. Fig. 28 provides a specification of knowledge source behavior in terms of three behavior trace assertions:

- If a knowledge source requires some subproblems P to be solved, in order to solve a problem p' and it gets solutions for all these subproblems q' on its input port cs, then it eventually solves the original problem p' and provides the solution through its output port ns (Eq. (15)).
- To solve a problem p, a knowledge source requires solutions only for smaller problems q (Eq. (16)).
- A knowledge source will eventually communicate its ability to solve an open problem via its output port rp (Eq. (17)).

3.3. Specifying Architectural Constraints

As the last step, a set of constraints about the activation and deactivation of components as well as constraints about connections between component ports are specified. Both types of constraints may be expressed in terms of *architecture trace assertions*, i.e, first-order linear temporal logic formulæ [MP92] over datatype variables (introduced in the description of behavior trace assertions above), component variables, and architecture assertions. A formal description of the syntax and semantics of architecture trace assertions is provided in App. B.

ASpec Name		**for** ifSpec
flex	aDt1:	`Sort1`
	aCmp1:	*If1*
rig	aDt2:	`Sort1`
	aCmp2:	*If1*
assertion1 (aDt1, aDt2, aCmp1, aCmp2)		
assertion2 (aDt1, aDt2, aCmp1, aCmp2)		

Fig. 29. Exemplary architecture constraint specification.

3.3.1. Component Variables

Component variables are typed by component types and may be interpreted by corresponding components. Similar to datatype variables, component variables can be classified into *flexible* and *rigid*, depending on whether they are newly interpreted at each point in time or whether they keep their value. Since component types may be parameterized, variables are assumed to be available for each possible valuation of the parameters. For example, a component variable declaration x for component type $X\langle bool\rangle$ would induce two component variables $x_{\langle true\rangle}$ and $x_{\langle false\rangle}$, which can be interpreted by components where parameter *bool* is valuated with the interpretations of *true* and *false*, respectively. Note that such parameterized component variables are only feasible since the semantics of a FACTum specification requires the existence of at least one component for each different valuation of a component type's parameters (as discussed in Sect. 3.2.1).

3.3.2. Architecture Assertions

Architecture assertions are predicates to specify snapshots of an architecture during execution (as described in Def. 4). Roughly speaking, they are predicate-logic formulæ specified over datatype and component variables, with terms consisting of:

- Datatype variables as well as component ports and component parameters, i.e., ports or parameters combined with corresponding component variables.
- Function and predicate symbols of the corresponding data type specification.

Moreover, several pre-defined, *architectural predicates* may be used for the formulation of architecture assertions:

- $c = c'$ denotes equality of two components c and c',
- $\widehat{c.p}$ denotes that a component c is currently sending/receiving a message over port p,
- $\wr c \wr$ denotes that a component c is currently active, and
- $c.p \rightsquigarrow c'.p'$ denotes that input port p of component c is connected to output port p' of component c'.

3.3.3. Architecture Constraint Specification

Architectural constraints can be specified by means of specification templates (Fig. 29). Each template has a name and is based on a corresponding interface specification. Then, a list of flexible and rigid variables for the data types and components are defined. Finally, a list of architecture trace assertions is formulated over the variables. Note that the semantics of architecture constraint specifications is given in terms of architecture specifications as described in Chap. 2. Thus, they can only be used to specify component activation and reconfiguration and *not* to restrict the behavior of components.

In the following, we provide architecture constraint specifications for all three running examples.

Example 20 (Architectural constraints for the Singleton pattern). Architectural constraints for the Singleton pattern are formalized by the specification depicted in Fig. 30. The specification requires two constraints for the activation of components: Eq. 18 requires that at each point in time there exists a singleton component c which is activated. Eq. 19 further asserts that there exists a unique component c', such that every active component c of type singleton is equal to c' at every point in time. In our version of the singleton, we require that the singleton component is not allowed to change over time. This is why

ASpec Singleton	**for** Singleton
flex c:	*Singleton*
rig c':	*Singleton*
$\Box\big(\exists c\colon \wr c \wr\big)$	(18)
$\exists c'\colon \Big(\Box\big(\forall c\colon (\wr c \wr \longrightarrow c = c')\big)\Big)$	(19)

Fig. 30. Activation constraints for a Singleton pattern.

ASpec Publisher-Subscriber	**for** Publisher-Subscriber
flex s:	*Subscriber*
p:	*Publisher*
m:	`msg`
E:	$\wp(\texttt{evt})$
rig s':	*Subscriber*
e:	`evt`
$\Box\Big(\wr p \wr \land \wr s \wr \land \widehat{s.sb} \longrightarrow p.sb \rightsquigarrow s.sb\Big)$	(20)
$\Box\Bigg(\wr s' \wr \land \big(\exists E\colon \text{sub } s' \ E \in s'.sb \land e \in E\big) \longrightarrow \Big(\big(\wr p \wr \land \wr s' \wr \land (e,m) \in p.nt \longrightarrow s'.nt \rightsquigarrow p.nt\big) \ \mathcal{W} \ \big(\wr s' \wr \land (\exists E\colon \text{unsub } s' \ E \in s'.sb \land e \in E)\big)\Big)\Bigg)$	(21)

Fig. 31. Architectural constraints for the Publisher-Subscriber pattern (in addition to the ones specified for the Singleton pattern).

variable c' is declared to be rigid in Fig. 30. Indeed, other versions of the singleton are possible in which the singleton may change over time.

Example 21 (Architectural constraints for the Publisher-Subscriber pattern). Activation constraints for the publisher component of a Publisher-Subscriber pattern are similar to the ones specified for the Singleton pattern in Fig. 30. Moreover, a Publisher-Subscriber pattern requires two additional constraints, regarding the connections between publisher and subscriber components, which are specified in Fig. 31:

- With Eq. (20), we require that a publishers *sb* port is always connected to a subscribers *sb* port, whenever both of them are active.
- With Eq. (21), we require that port *nt* of a subscriber s' is always connected to a publishers *nt* port, whenever the publisher sends out a message associated to an event e for which s' is subscribed.

Example 22 (Architectural constraints for the Blackboard pattern). Also for the Blackboard pattern, we get similar activation constraints for blackboard components as the ones specified for the Singleton pattern in Fig. 30. Moreover, the Blackboard pattern requires similar connection constraints as required for the Publisher-Subscriber pattern in Fig. 31. Thereby, port *rp* of the Blackboard pattern corresponds to port *sb* and port *cs* of the Blackboard pattern to port *nt*.

In addition, Fig. 32 provides two connection constraints and three activation constraints for Blackboard architectures:

- By Eq. (22), we require that a blackboard's *op* port is always connected to a knowledge source's *op* port, whenever both of them are active.

ASpec Blackboard		**for** Blackboard
flex	ks :	$KS\langle prob \rangle$
	bb :	BB
	p :	PROB
	s :	SOL
	P :	$\wp$(PROB)
rig	ks' :	$KS\langle prob \rangle$
	p' :	PROB

$$\Box\Big(\wr ks \wr \land \wr bb \wr \land \widehat{bb.op} \longrightarrow ks.op \leadsto bb.op\Big) \tag{22}$$

$$\Box\Big(\wr bb \wr \land \wr ks \wr \land \widehat{ks.ns} \longrightarrow bb.ns \leadsto ks.ns\Big) \tag{23}$$

$$\Box\Big(\wr ks' \wr \longrightarrow \Box\big(\Diamond \wr ks' \wr\big)\Big) \tag{24}$$

$$\Box\Big(\wr ks' \wr \land (p, P) \in ks'.rp \land p' \in P \longrightarrow \Box\big((\exists bb\colon \wr bb \wr \land (p', s) \in bb.cs) \longrightarrow \wr ks' \wr\big)\Big) \tag{25}$$

$$\Box\Big(\forall p' \in bb.op\colon \Diamond\big(\exists ks_{\langle prob=p' \rangle}\colon \wr ks \wr\big)\Big) \tag{26}$$

Fig. 32. Specification of constraints for Blackboard architectures.

- By Eq. (23), we require that a blackboard's *ns* port is always connected to a knowledge source's *ns* port, whenever both of them are active.
- By Eq. (24), we require a fairness condition for the activation of already activated knowledge sources.
- By Eq. (25), we require that whenever a knowledge source offers to solve some problem p, given that it receives solutions for corresponding subproblems P, then the knowledge source is activated, whenever solutions for any of the problems of P are provided.
- By Eq. (26), we require that for each open problem p', a knowledge source ks which is able to solve p' is eventually activated.

Note expression $\exists ks_{\langle prob=p' \rangle}\colon \wr ks \wr$ of Eq. (26) which demonstrates the use of parameterized component variables. Indeed, the variable $ks_{\langle prob=p' \rangle}$ represents a knowledge source component which has its parameter "*prob*" valuated with the problem represented by datatype variable p'. Such parameterized variables provide a convenient way to specify constraints about components of parameterized component types.

3.4. Architecture Diagrams

So far, we introduced basic techniques to specify ADPs. We also applied the techniques to specify versions of three, well-known, ADPs: the Singleton, the Publisher-Subscriber, and the Blackboard pattern.

Specifying these patterns, however, led to two further observations:

1. Some architectural constraints are common to different ADPs. One example is the constraint that components of a certain type are required to be always activated. Another example is that components of a certain type are connected via certain ports, whenever they are activated.

2. Another observation is that, sometimes, pattern specifications reuse specifications from other patterns. For example, the specification of the Publisher-Subscriber pattern reused the activation specification from the Singleton pattern for the publisher component. Or the Blackboard pattern reused the complete specification of the Publisher-Subscriber pattern.

Building on these observations, in the following section, we extend our specification approach with two features which turn out to be useful for the specification of patterns:

1. To facilitate the specification of common activation and connection constraints, we extend our notion of architecture diagram (introduced for the specification of interfaces in Sect. 3.2.1) with so-called *activa-*

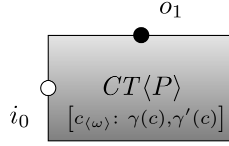


Fig. 33. Activation annotation for a component type $CT\langle P\rangle$ with minimal activation condition $\gamma(c)$ and deactivation condition $\gamma'(c)$.

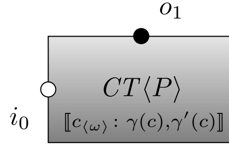


Fig. 34. Rigid activation annotation for a component type $CT\langle P\rangle$, with activation condition $\gamma(c)$ and deactivation condition $\gamma'(c)$.

tion/connection annotations. These annotations provide a convenient way to express certain architectural constraints graphically by annotating the given architecture diagram.

2. To support hierarchical pattern specifications, we introduce the notion of *pattern instantiations*, which allow to import a pattern specification from within another pattern specification and instantiate the corresponding component types. Therefore, we provide additional annotations for architecture diagrams, which allow to easily express such instantiations in a graphical manner.

Again, the different techniques are demonstrated in terms of our three running examples.

3.4.1. Activation Annotations

Activation annotations enhance an architecture diagram with constraints regarding the activation and deactivation of components. They are expressed by annotating component types with corresponding architecture assertions, determining situations in which components of the annotated type are required to be activated or deactivated. Fig. 33, for example, depicts an activation annotation for a component type CT, parameterized by a parameter P. The annotation is enclosed between square brackets and takes a variable c of a component of type CT, with parameter valuation ω, as input. It then specifies two architecture assertions using variable c: $\gamma(c)$ determines situations in which the component is required to be activated, while $\gamma'(c)$ determines a situation in which the component is required to be deactivated. For the case neither $\gamma(c)$ nor $\gamma'(c)$ holds, c may be either active or not. If omitted, we assume default values *true* and *false*, respectively. To specify only the first parameter and leave the default value for the other one, we write $\lfloor c_{\langle\omega\rangle}: \gamma(c)\rfloor$. Similarly, we write $\lceil c_{\langle\omega\rangle}: \gamma(c)\rceil$ to only specify the second parameter and leave the default value for the other one.

Activation annotations, as described so far, specify the activation/deactivation of components of a certain type. However, they do not say anything about the identity of these components. The annotation in Fig. 33, for example, allows c to be a different component at different points in time. To require that the identity of the components does not change over time, we need a stronger notion of activation annotation. We call it *rigid activation annotation*, and it is expressed using double square brackets, instead of single square brackets. Fig. 34, for example, depicts an activation annotation, similar to the one presented in Fig. 33. However, since we use double square brackets, it is to be interpreted as a rigid activation annotation and variable c is not allowed to change over time. Similar as for activation annotations we take *true* and *false* as default values and write $\|\lfloor c\colon \gamma(c)\rfloor\|$ and $\|\lceil c\colon \gamma(c)\rceil\|$ to take the default values for the second and first condition only.

Using activation annotations, we can now specify the Singleton pattern more concisely by adapting the architecture diagram presented in Fig. 23.

Example 23 (Annotations for the Singleton pattern). Figure 35 depicts the adapted architecture diagram for Singletons. The first condition requires that a singleton is always activated. The second condition, on the other hand, requires that, whenever a singleton is activated, it is the only component of that type which is active. Since we do not want singletons to change over time, we enclose the conditions in double

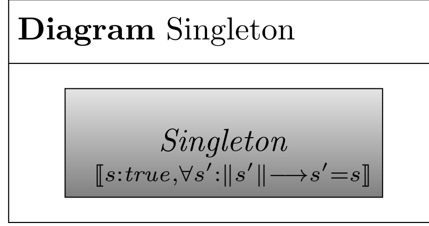


Fig. 35. Annotated architecture diagram for the Singleton pattern.

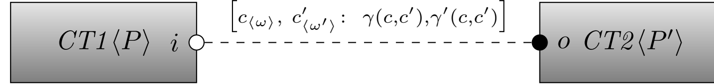


Fig. 36. Connection annotation for connections between port i of component type $CT1\langle P\rangle$ and port o of component type $CT2\langle P'\rangle$ with connection condition $\gamma(c,c')$ and disconnection condition $\gamma'(c,c')$.

square brackets, making it a rigid activation annotation. Note that the activation specification presented in Fig. 30 becomes now superfluous, since the new architecture diagram contains the complete specification of the Singleton pattern.

3.4.2. Connection Annotations

Connection annotations enhance an architecture diagram with constraints regarding the connection of certain components. Such annotations are added to each type-consistent pair of input and output ports of component types and specify conditions under which the corresponding ports of components of these types are required to be connected or disconnected. Figure 36, for example, depicts a connection annotation for ports i and o of component types *CT1* and *CT2*, respectively. The annotation takes two component variables, c and c', as input and specifies two architecture assertions, $\gamma(c,c')$ and $\gamma'(c,c')$, over these variables: $\gamma(c,c')$ determines situations in which the connection is required to be established, while $\gamma'(c,c')$ determines a situation in which a connection is not allowed. For the case neither $\gamma(c,c')$ nor $\gamma'(c,c')$ holds, the connection may be established or not. Again, we assume default values of *true* and *false*, and we may omit one of the conditions in order to take its default value. Also for connection annotations we may require components not to change over time, which is why we also introduce the notion of *rigid connection annotation*. Similar as for rigid activation annotations, we mark connection annotations as rigid by enclosing them into double square brackets, instead of single square brackets.

In the following, we demonstrate the use of connection annotations in terms of the examples introduced above.

Example 24 (Annotations for the Publisher-Subscriber pattern). We first adapt the architecture diagram for the Publisher-Subscriber pattern introduced in Fig. 24. The resulting architecture diagram is depicted in Fig. 37. We require a similar activation annotation for a publisher component as the one introduced for singletons. To increase readability, however, the annotation uses abbreviations γ and γ', which are expanded at the bottom of the diagram. Moreover, we add a connection annotation, which requires port *sb* of a publisher component to be connected to port *sb* of a subscriber component, whenever the subscriber sends out some message. The dashed line, without any annotation, denotes a connection constraint using the default values for connections and deconnections. Indeed the line could have been omitted and the semantic would not change. However, it is put there to highlight the fact that there is an additional connection constraint specified as architecture trace assertion in Eq. (21). The new architecture diagram allows us now to get rid of some of the architectural assertions introduced in Ex. 21. Indeed, the only remaining assertion is Eq. (21), which cannot be replaced with any annotation so far. Note that the connection annotation used in this example is used frequently which is why, from now one, we shall use a solid connection between ports to denote that the corresponding ports are required to be connected, whenever the output port is valuated with some message.

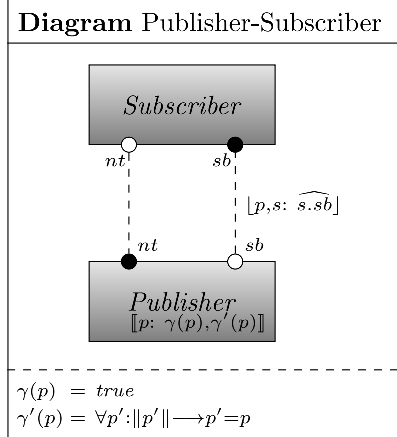


Fig. 37. Annotated architecture diagram for the Publisher-Subscriber pattern.

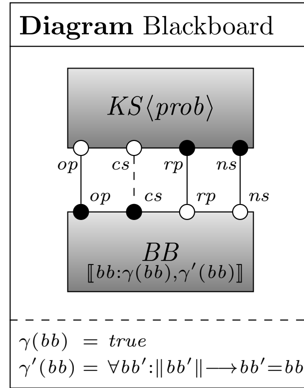


Fig. 38. Annotated architecture diagram for the Blackboard pattern.

Example 25 (Annotations for the Blackboard pattern). Next, we adapt the architecture diagram for the Blackboard pattern as introduced in Fig. 25. The resulting architecture diagram is depicted in Fig. 38. Again, we add a similar annotation as the one required for singletons to the blackboard component type. Moreover, we add three connection annotations: the three solid lines between the ports of blackboard and knowledge source components use the new notation, introduced in the last example, to denote a required connection between the corresponding ports, whenever the output port sends out a message. The architecture diagram now subsumes the two connection assertions Eq. (22) and Eq. (23) specified for Blackboard architectures in Fig. 32. What remains are the activation assertions Eq. (24), Eq. (25), and Eq. (26), as well as the connection assertion formulated by Eq. (21).

3.4.3. Specifying Pattern Instantiations

As described above, pattern specifications may be built on top of other pattern specifications by instantiating their component types. Instantiating a pattern requires to provide a mapping which relates component types and port types. Such instantiations can be directly specified in a pattern's architecture diagram by annotating the corresponding interfaces.

Figure 39 depicts a schematic pattern instantiation. The diagram specifies a pattern *PatternB* and thereby instantiates another, existing specification *PatternA*, which is assumed to specify a component type *CT1* with one single input port i and one single output port o. *PatternB* specifies one component type *CT2* which is declared to be an instance of component type *CT1* of *PatternA*. Since *CT1* has two ports, the instantiation must provide mappings for these two ports, which is done in square brackets right after the

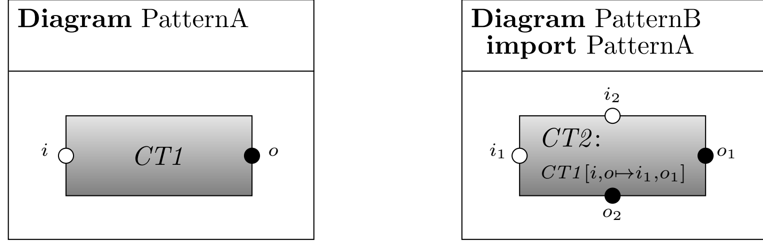


Fig. 39. Architecture diagram in which component type *CT1* of *PatternB* instantiates component type *CT1* of a pattern *PatternA*.

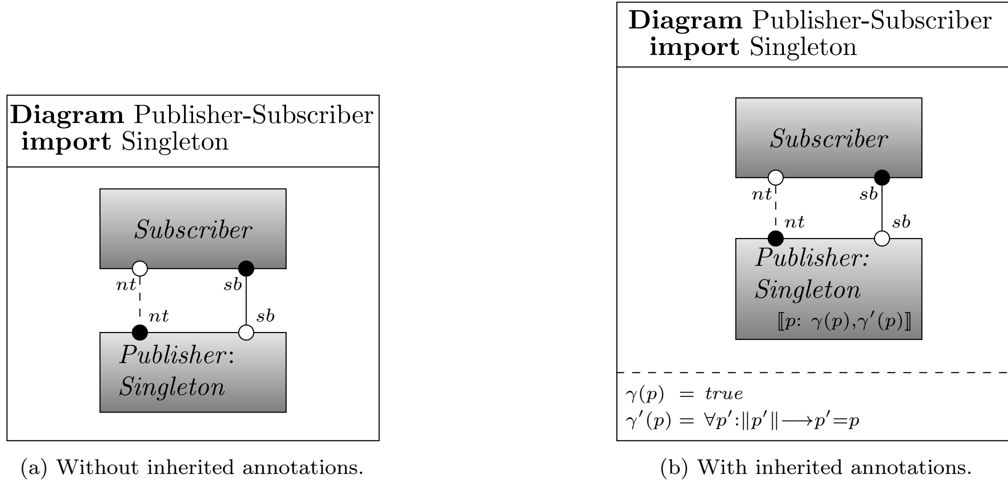


Fig. 40. Architecture diagrams for the Publisher-Subscriber pattern instantiating the Singleton pattern.

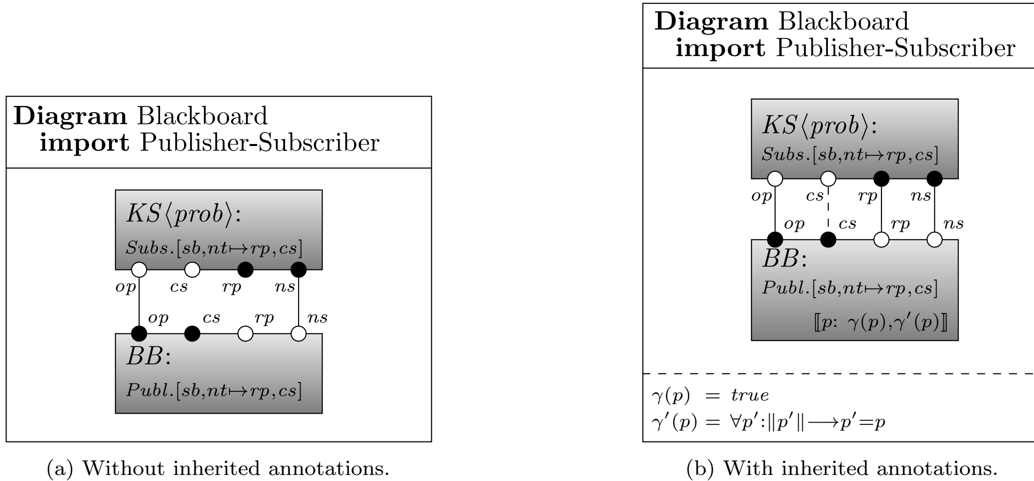
name of the instantiated component type. In our case, port i of component type *CT1* is mapped to input port i_1 and o to output port o_1. Note that the interface of *CT2* has two additional ports i_2 and o_2, which do not instantiate any port of interface *CT1*. Indeed, a port mapping is *not* required to be bijective, which means that a component type may add more ports to the interface of the component type it instantiates. However, we do require that the types of the ports refine the types of the port of the instantiated component type. For our example, that means the type of port i_1 must be a refined version of the type of i and the type of o_1 must refine the type of o.

In the following, we demonstrate hierarchical specifications in terms of our running examples.

Example 26 (Publisher-Subscriber instantiating the Singleton pattern). First, we adapt the specification of the Publisher-Subscriber pattern, introduced above, such that a publisher component is considered to be an instance of the singleton type. Figure 40 depicts the adapted architecture diagrams, excluding (Fig. 40a) and including (Fig. 40b) annotations inherited from the imported Singleton pattern. The diagram first imports the specification of the Singleton. Then it declares the publisher component type to be an instance of a singleton component type from the Singleton pattern.

By instantiating the singleton, the publisher will inherit all its specified properties, i.e., an adapted version of the activation annotation specified for the Singleton will be available for publisher components. Thus, we could simplify the remaining assertion for Publisher-Subscriber patterns and we are left with a simplified version of Eq. (21) for the final specification of the pattern (in addition to the diagram depicted in Fig. 40).

Example 27 (Blackboard pattern instantiating the Publisher-Subscriber pattern). Finally, we model a Blackboard pattern as an instance of the Publisher-Subscriber pattern. Thereby, the blackboard is specified to be an instance of the publisher type, and knowledge sources instances of subscriber components, respectively. Figure 41 depicts the adapted architecture diagrams, excluding (Fig. 41a) and including (Fig. 41b) annotations inherited from the imported Publisher-Subscriber pattern. Again, the diagram im-



(a) Without inherited annotations.
(b) With inherited annotations.

Fig. 41. Architecture diagrams for the Blackboard pattern instantiating the Publisher-Subscriber pattern.

ports the specification of the Publisher-Subscriber pattern and declares a blackboard to be an instance of a publisher and a knowledge source to be an instance of a subscriber. However, since publisher as well as subscriber interfaces have ports, we need to provide an additional port-mapping which maps every port of a publisher/subscriber to a corresponding port of a blackboard/knowledge source. Note also that a blackboard/knowledge source adds two additional ports which do not map to any of the ports of the Publisher-Subscriber pattern.

Again, by instantiating the Publisher-Subscriber pattern, the Blackboard will inherit the specification of the Publisher-Subscriber pattern. Thereby, the properties are adapted as specified by the instantiation, i.e., publisher components will become blackboard components and subscriber components will become knowledge sources in the specified properties. In addition, inherited properties will be propagated as well, which means that the Blackboard will even inherit the properties of the Singleton, since the publisher component instantiates the Singleton. Thus, the final specification of the Blackboard pattern will only require three activation assertions (in addition to the diagram depicted in Fig. 41), specified by Eq. (24), Eq. (25), and Eq. (26). Note that the specification of these properties (as provided in Fig. 32) allows bb to be flexible which, at a first glance, might seem to contradict the requirement of uniqueness of a blackboard component inherited from the Singleton. However, while flexible variables may indeed be interpreted by different components over time, these components are not required to be different. Thus, for a model in which only one component of a certain type exists, a flexible variable of that type will always be interpreted by that component.

3.5. Summary

Table 1 summarizes techniques for the specification of ADPs described in this section. For each technique, it lists the specified elements, the corresponding architectural concept, and the type of specification technique.

4. Interactive Pattern Verification in Isabelle/HOL

In the previous sections, we presented a model for dynamic architectures (Sect. 2) and techniques to specify ADPs over this model (Sect. 3). ADPs can then be verified by proving guarantees for them from such specifications. Until now, however, verification relies on traditional "pencil and paper" proofs, which are not guaranteed to be sound. To address this problem, we integrated our approach with the interactive theorem prover Isabelle/HOL [NPW02].

The integration is shown in Fig. 42: First, we formalized our model for dynamic architectures (presented in Sect. 2) in Isabelle/HOL. The corresponding theory `Configuration_Traces.thy` is available in the Archive of Formal Proofs as entry *DynamicArchitectures* [Mar17a]. It uses Lochbihler's formalization of coinductive

	Specification Elements	*Concepts from Sect. 2*	*Type*
Algebraic Data Types	Datatype Variables Axioms Generator Clauses	Messages (Sect. 2.1)	Template
Port Specifications	Port Types	Ports and Type Function (Sect. 2.1)	Template
Architecture Diagrams	Interfaces	Interfaces (Sect. 2.3)	Graphical
Activation Annotations	Upper Bounds Lower Bounds	Component Activation (Sect. 2.6)	Annotation
Connection Annotations	Upper Bounds Lower Bounds	Port Connections (Sect. 2.6)	Annotation
Behavior Trace Assertions	Datatype Variables* Behavior Assertions	Component Types (Sect. 2.5)	Template
Architecture Trace Assertions	Datatype Variables* Component Variables* Architecture Assertions	Architecture Specification (Sect. 2.6)	Template

* flexible vs. rigid

Table 1. Summary of specification techniques for ADPs.

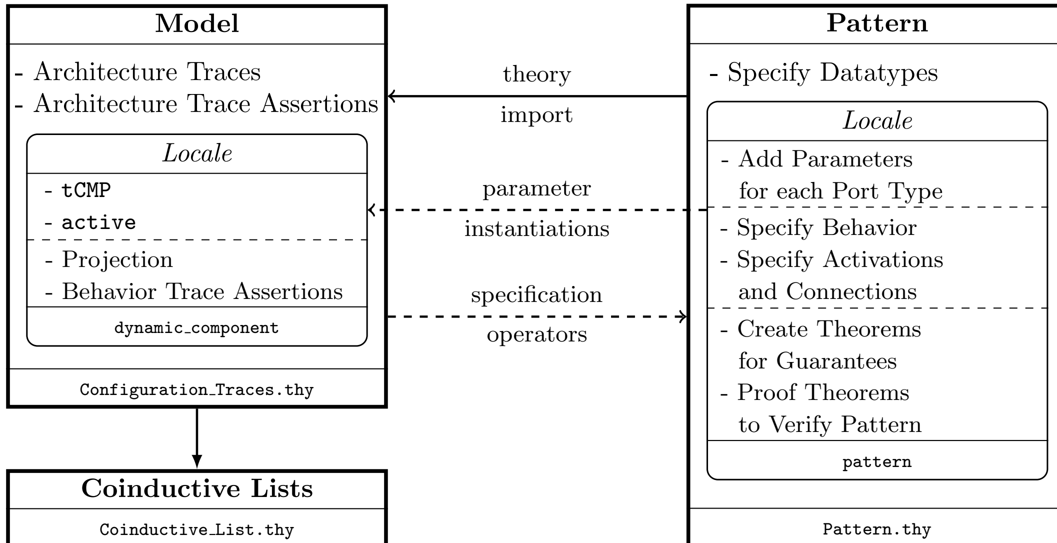


Fig. 42. Overview of verification approach.

lists [Loc10] to formalize the notion of architecture traces and architecture trace assertions. Moreover, it introduces an Isabelle locale [Bal04] `dynamic_component`, which requires two parameters: a function *tCMP*, to extract a snapshot of a component from an architecture snapshot, and a function *active*, to assert whether a certain component is active in an architecture snapshot. Then, it introduces several definitions for the locale, reflecting the definitions presented in Sect. 2. To verify a pattern, one may create an Isabelle/HOL theory which imports our model. The pattern theory contains Isabelle datatype specifications for each FACTum datatype specification. Moreover, it contains an Isabelle locale with parameters for all port types of all component types involved in the pattern and assumptions for all constraints imposed by the pattern. Finally, a pattern guarantee is specified in terms of an Isabelle/HOL theorem and proved from the assumptions using Isabelle's structured proof language Isabelle/Isar [Wen07].

In the following, we detail on each of the steps of our verification approach: First, we describe the formalization of our model in more detail. Then, we show how a FACTum specification can be mapped to a corresponding Isabelle/HOL theory over the model and demonstrate it in terms of our three running examples. Finally, we discuss how a FACTum specification can be verified in Isabelle/HOL and demonstrate it in terms of our examples.

```
codatatype (lset: 'a) llist =
    lnull: LNil
  | LCons (lhd: 'a) (ltl: 'a llist)
for
  map: lmap
  rel: llist-all2
where
  lhd LNil = undefined
| ltl LNil = LNil
```

Fig. 43. Formalization of lazy lists in Isabelle/HOL (excerpt from [Loc10]).

4.1. Formalizing Architecture Traces

Since our formalization is based on coinductive lists, we first summarize their formalization in Isabelle. Then, we demonstrate our formalization of the model presented in Sect. 2 using coinductive lists.

4.1.1. Coinductive Lists

To deal with possibly infinite architecture traces, our formalization is based on Lochbihler's theory of coinductive (lazy) lists [Loc10]. Lazy lists are formalized using Isabelle/HOL's notion of coinductive datatypes [BHL+14]. Figure 43 depicts the corresponding Isabelle/HOL fragment: Besides introducing the codatatype itself, the declaration also introduces some auxiliary constants:

- Destructors *LNil* and *LCons*.
- Discriminator *lnull*, to test whether a list is empty.
- Selectors *lhd* and *ltl*, to select the first element of a given list and the remaining tail, respectively.
- Set function *lset*, which returns a (possibly infinite) set containing all the elements of a given list.
- Map function *map*, to apply a given function to each element of a certain list.
- Relator *rel*, to compare two lists based on their elements.

The where clause at the end of the command specifies a default value for selectors *lhd* and *ltl*, applied to *LNil*, on which they are not a priori specified.

In addition, Lochbihler's theory introduces several definitions for lazy lists, of which the following are most relevant for our purpose:

inf-llist converts a function with domain of natural numbers to a corresponding infinite list.
llength returns the (possibly infinite) length of a list.
lnth returns the n-th element of a list.
lappend concatenates two lists.
lfilter extracts a sublist which contains only elements characterized by a given predicate.
ltake returns a prefix of a certain length of a given list.

Since *lfilter* and *ltake* are of particular importance to our theory, we discuss them in more detail.

Lazy filter function. The *lfilter* function is important, since it forms the foundation for our formalization of the behavior projection operator. The function takes a predicate P and a lazy list xs and returns a sublist, containing only those elements of xs for which P holds. Its definition is provided in Fig. 44: It is formalized as a recursive function based on fixpoints in complete-partial-orders. Note that the definition does not require any termination proof (which would indeed not be possible, in general, since we are talking about infinite lists). Rather, to guarantee the existence of a fixpoint, the definition must ensure that the induced functional is monotonic w.r.t. the prefix order for lazy lists.

Lazy take function. Another important function is the *ltake* function, since it is used to formalize our notion of number of activations of a component in a architecture trace. The function takes an *extended* natural number n (including ∞) and a lazy list xs, and returns a sublist, containing the first n elements of

```
partial-function (llist) lfilter :: 'a llist ⇒ 'a llist
where lfilter xs = (case xs of LNil ⇒ LNil
                   | LCons x xs' ⇒ if P x then LCons x (lfilter xs') else lfilter xs')
```

Fig. 44. Formalization of lazy filter function in Isabelle/HOL (excerpt from [Loc10]).

```
primcorec ltake :: enat ⇒ 'a llist ⇒ 'a llist
where
  n = 0 ∨ lnull xs ⟹ lnull (ltake n xs)
| lhd (ltake n xs) = lhd xs

| ltl (ltake n xs) = ltake (epred n) (ltl xs)
```

Fig. 45. Formalization of lazy take function in Isabelle/HOL (excerpt from [Loc10]).

xs. Figure 45 depicts its formalization in Isabelle/HOL: It is formalized as a primitive corecursive function, in which the syntactic structure of the definition ensures productivity (and thus well-definedness) of the function.

4.1.2. Formalizing Architecture Traces

In the following, we describe a possible formalization of the model presented in Sect. 2 using coinductive lists. The following Isabelle/HOL snippet depicts the foundation of our formalization:

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

First, we introduce a type constant *cnf*, which represents an architecture snapshot, i.e, the state of an architecture during system execution. An architecture trace is then formalized as a function which assigns a snapshot *cnf* to each point in time *nat*. Finally, an architecture *arch* is modeled as a set "*trace set*" of architecture traces.

As mentioned above, the interface to the model is given in terms of an Isabelle/Locale [Bal04]:

```
locale dynamic-component =
 fixes tCMP :: 'id ⇒ cnf ⇒ 'cmp (σ_(_) [0,110]60)
  and active :: 'id ⇒ cnf ⇒ bool (‖_‖_ [0,110]60)
```

The locale introduces two type parameters:
'id represents a type containing component identifiers.
'cmp represents a type containing component snapshots.
Moreover, it requires two function parameters:
tCMP is an operator to extract the state of a component with a certain identifier *'id* from an architecture snapshot *cnf*.
active is a predicate to assert whether a component with a certain identifier *'id* is activated within an architecture snapshot *cnf*.
The locale introduces several operators for architecture traces, along with some characteristic properties thereof.

Component projection. Perhaps the most important operator is behavior projection introduced by Def. 7. As discussed in Sect. 2.7.1, the operator takes a component identifier c and an architecture trace t, and returns a so called *behavior trace*, i.e., a list containing all the states of component c in t. Thereby, all the time points in which component c is not activated in t are removed. The operator is formalized by combining the lazy filter function *lfilter* (described above) with the lazy map function *lmap*:

```
definition proj:: 'id ⇒ (cnf llist) ⇒ ('cmp llist) (π_(_) [0,110]60)
 where proj c = lmap (λcnf. (σ_c(cnf))) ∘ (lfilter (active c))
```

First, *lfilter* is used to remove all time points in t, for which c is not activated (using locale parameter

active). Then, *lmap* is used to extract the state of a component out of a given architecture snapshot (using locale parameter *tCMP*).

Least deactivation. The least deactivation operator was first introduced in Def. 5 and it is formalized using Isabelle/HOL's definite description operator *LEAST*, which returns the *least* element which satisfies a certain condition (or an arbitrary element of the corresponding type if no element satisfied the condition):

```
definition lNAct :: 'id ⇒ (nat ⇒ cnf) ⇒ nat ⇒ nat (⟨- ⇐ -⟩-)
 where ⟨c ⇐ t⟩n ≡ (LEAST n'. n=n' ∨ (n'<n ∧ (∄k. k≥n' ∧ k<n ∧ ‖c‖t k)))
```

It takes a component identifier c, an architecture trace t, and a time point n and returns the time point right after the last activation of c in t prior to n.

Next activation. As described in Def. 5, the next operator takes a component identifier c, an architecture trace t, and a time point n, and returns the next point in time (including n) at which c is active in t.

```
definition nxtAct :: 'id ⇒ (nat ⇒ cnf) ⇒ nat ⇒ nat (⟨- → -⟩-)
 where ⟨c → t⟩n ≡ (THE n'. n'≥n ∧ ‖c‖t n' ∧ (∄k. k≥n ∧ k<n' ∧ ‖c‖t k))
```

Note that the formalization of the next operator uses Isabelle's definite description operator *THE*, which returns the *unique* element which satisfies a given condition if such an element exists (or an arbitrary element of the corresponding type if no such element exists).

Latest activation. The latest activation operator was introduced in Def. 5 and returns the latest activation of a component *before* a certain point in time. Again, its formalization uses one of Isabelle/HOL's definite description operator:

```
definition latestAct :: 'id ⇒ trace ⇒ nat ⇒ nat (⟨- ← -⟩-)
 where ⟨c ← t⟩n = (GREATEST n'. n'<n ∧ ‖c‖t n')
```

Note that *GREATEST* in Isabelle/HOL is the dual of *LEAST*. It returns the *greatest* element which satisfies a certain condition or an arbitrary element of the corresponding type if element satisfied the condition.

Last activation. The last point in time at which a component is active in an architecture trace can be obtained using operator *lActive*, introduced in Def. 5. It is again formalized using Isabelle/HOL's *GREATEST* operator:

```
definition lActive :: 'id ⇒ (nat ⇒ cnf) ⇒ nat (⟨- ∧ -⟩)
 where ⟨c ∧ t⟩ ≡ (GREATEST i. ‖c‖t i)
```

4.1.3. Specifying Architecture Traces

To specify architecture traces, we formalized our notion of architecture trace assertion (introduced in Sect. 3). To this end, we first introduced a type synonym for architecture trace assertions:

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

Then, we introduced a mapping to lift an architecture assertion to a corresponding architecture trace assertion:

```
definition ca :: (cnf ⇒ bool) ⇒ cta
  where ca φ ≡ λ t n. φ (t n)
```

Finally, we defined each of the operators used in the specification of architecture trace assertions in terms of predicate transformers, i.e., functions which take an architecture trace assertion and modify it accordingly.

Logical connectives. First, we introduced definitions for the basic logical operators:

```
definition neg :: cta ⇒ cta (¬c - [19] 19)
  where ¬c γ ≡ λ t n. ¬ γ t n
definition conj :: cta ⇒ cta ⇒ cta (infixl ∧c 20)
```

```
  where γ ∧^c γ' ≡ λ t n. γ t n ∧ γ' t n
definition disj :: cta ⇒ cta ⇒ cta (infixl ∨^c 15)
  where γ ∨^c γ' ≡ λ t n. γ t n ∨ γ' t n
definition imp :: cta ⇒ cta ⇒ cta (infixl ⟶^c 10)
  where γ ⟶^c γ' ≡ λ t n. γ t n ⟶ γ' t n
```

They mainly lift each corresponding HOL operator to the level of architecture traces. In a similar way, we introduced quantifiers for architecture trace assertions:

```
definition all :: ('a ⇒ cta) ⇒ cta (binder ∀_c 10)
  where all P ≡ λt n. (∀y. (P y t n))
definition ex :: ('a ⇒ cta) ⇒ cta (binder ∃_c 10)
  where ex P ≡ λt n. (∃y. (P y t n))
```

Temporal operators. Then, we introduced definitions for each temporal logic operator. Their semantics indeed resembles the traditional semantics of linear temporal logics [MP92].

Temporal logic *next* is implemented as a function which takes an architecture trace assertion γ and returns another architecture trace assertion which evaluates γ at the next point in time.

```
definition nxt :: cta ⇒ cta (○_c(-) 24)
  where ○_c(γ) ≡ λt n. γ t (Suc n)
```

Eventually is formalized as a function which takes an architecture trace assertion γ and returns another architecture trace assertion which evaluates γ somewhere in the future:

```
definition evt :: cta ⇒ cta (◇_c(-) 23)
  where ◇_c(γ) ≡ λt n. ∃n'≥n. γ t n'
```

The *globally* operator transforms an architecture trace assertion γ to another architecture trace assertion which evaluates γ at every time in the future:

```
definition glob :: cta ⇒ cta (□_c(-) 22)
  where □_c(γ) ≡ λt n. ∀n'≥n. γ t n'
```

Finally, *until* takes two architecture trace assertions, γ and γ', and evaluates γ' in the future as long as γ does not hold:

```
definition until :: cta ⇒ cta ⇒ cta (infixl 𝔘_c 21)
  where γ' 𝔘_c γ ≡ λt n. ∃n''≥n. γ t n'' ∧ (∀n'≥n. n' < n'' ⟶ γ' t n')
```

We also introduce a *weaker* notion of *until* as a combination of the traditional until and globally:

```
definition wuntil :: cta ⇒ cta ⇒ cta (infixl 𝔚_c 20)
  where γ' 𝔚_c γ ≡ γ' 𝔘_c γ ∨^c □_c(γ')
```

4.1.4. Specifying Component Behavior

As described in Sect. 3, component behavior is specified using behavior trace assertions. Just as for architecture trace assertions, we start the formalization by introducing a corresponding type synonym:

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

Thereby, a behavior trace assertion is formalized as a predicate over a behavior trace and a natural number. Here, the state of a component is modeled in terms of a type parameter $'c$.

Similar as for architecture trace assertions, we then introduced an operator to lift behavior assertions to corresponding behavior trace assertions:

```
definition pred :: bool ⇒ ('cmp bta)
  where pred P ≡ λ t n. P
```

In addition, we also introduce an operator to lift an arbitrary HOL predicate to a corresponding behavior trace assertion:

```
definition ba :: ('cmp ⇒ bool) ⇒ ('cmp bta)
  where ba φ ≡ λ t n. φ (t n)
```

Note that such a definition was not required for architecture trace assertions, since architecture assertions can be used to lift arbitrary predicates to the level of architecture trace assertion. For behavior assertions, this is not possible since they are evaluated only at time points at which a component is indeed active.

Similar as for architecture trace assertions, we defined each of the operators, used for the specification of behavior trace assertions, in terms of predicate transformers.

Logical connectives. Basic logical connectives are defined in a similar way as for architecture trace assertions:

```
definition imp :: ('cmp bta) ⇒ ('cmp bta) ⇒ ('cmp bta) (infixl ⟶^b 10)
  where γ ⟶^b γ' ≡ λ t n. γ t n ⟶ γ' t n
definition disj :: ('cmp bta) ⇒ ('cmp bta) ⇒ ('cmp bta) (infixl ∨^b 15)
  where γ ∨^b γ' ≡ λ t n. γ t n ∨ γ' t n
definition conj :: ('cmp bta) ⇒ ('cmp bta) ⇒ ('cmp bta) (infixl ∧^b 20)
  where γ ∧^b γ' ≡ λ t n. γ t n ∧ γ' t n
definition neg :: ('cmp bta) ⇒ ('cmp bta) (¬^b - [19] 19)
  where ¬^b γ ≡ λ t n. ¬ γ t n
```

Behavior assertions also support quantification over variables of a certain type. Again, Isabelle/HOL quantifiers are used to formalize quantification for behavior assertions:

```
definition all :: ('a ⇒ ('cmp bta)) ⇒ ('cmp bta) (binder ∀_b 10)
  where all P ≡ λt n. (∀y. (P y t n))
definition ex :: ('a ⇒ ('cmp bta)) ⇒ ('cmp bta) (binder ∃_b 10)
  where ex P ≡ λt n. (∃y. (P y t n))
```

Temporal operators. Similar as for architecture trace assertions, we formalize temporal operators for behavior trace assertions using their traditional semantics [MP92]:

```
definition nxt :: ('cmp bta) ⇒ ('cmp bta) (○_b(-) 24)
  where ○_b(γ) ≡ λ t n. γ t (Suc n)
definition evt :: ('cmp bta) ⇒ ('cmp bta) (◇_b(-) 23)
  where ◇_b(γ) ≡ λ t n. ∃n'≥n. γ t n'
definition glob :: ('cmp bta) ⇒ ('cmp bta) (□_b(-) 22)
  where □_b(γ) ≡ λ t n. ∀n'≥n. γ t n'
definition until :: ('cmp bta) ⇒ ('cmp bta) ⇒ ('cmp bta) (infixl 𝔘_b 21)
  where γ' 𝔘_b γ ≡ λ t n. ∃n''≥n. γ t n'' ∧ (∀n'≥n. n' < n'' ⟶ γ' t n')
definition wuntil :: ('cmp bta) ⇒ ('cmp bta) ⇒ ('cmp bta) (infixl 𝔚_b 20)
  where γ' 𝔚_b γ ≡ γ' 𝔘_b γ ∨^b □_b(γ')
```

4.1.5. Formalizing Composition

As explained in Def. 8, behavior specifications are composed with architecture specifications using behavior projection. To formalize composition, we introduced an evaluation function which takes a specification of a component behavior (not considering component activation and deactivation) and interprets it over an architecture trace (in which the component is subject to activation and deactivation). First, however, we needed two additional operators to obtain a component's number of activations and to map time points between components and architectures.

Number of activations. This operator returns the number of activations of a certain component within a given architecture trace. Intuitively, the operator takes a component identifier c, a time point n, and an architecture trace t and returns the number of activations of c up to (and including) time point n. The operator is formalized by combining component projection with the lazy take function *ltake* and the lazy length function (both described above in Sect. 4.1.1):

```
definition nAct :: 'id ⇒ enat ⇒ (cnf llist) ⇒ enat (⟨- #_-⟩)
  where ⟨c #_n t⟩ ≡ llength (π_c(ltake n t))
```

First, *ltake* is used to obtain a sublist of length n of the original architecture trace t. Then, component projection is applied to the remaining architecture trace to remove all time points in which component c is not active. What is left is a lazy list containing all the activations of c in t up to time point n and we simply return its length.

Mapping time points. Applying behavior projection for a component c to an architecture trace t, results in a behavior trace which contains all the states of c, whenever it is active in t. Thereby, the time points at which a certain state of c is available, after applying projection, may change (due to the deactivation of c in t). Thus, to map time points in between an architecture trace and the corresponding projection, we introduce two additional operators:

definition $\mathit{cnf2bhv} :: {'id} \Rightarrow (nat \Rightarrow cnf) \Rightarrow nat \Rightarrow nat$ (-↓-(-) [150,150,150] 110)
 where $_{c}{\downarrow}_{t}(n) \equiv \mathit{the\text{-}enat}(\mathit{llength}\ (\pi_{c}(\mathit{inf\text{-}llist}\ t))) - 1 + (n - \langle c \wedge t\rangle)$
definition $\mathit{bhv2cnf} :: {'id} \Rightarrow (nat \Rightarrow cnf) \Rightarrow nat \Rightarrow nat$ (-↑-(-) [150,150,150] 110)
 where $_{c}{\uparrow}_{t}(n) \equiv \langle c \wedge t\rangle + (n - (\mathit{the\text{-}enat}(\mathit{llength}\ (\pi_{c}(\mathit{inf\text{-}llist}\ t))) - 1))$

Note that *cnf2bhv* is used to map a given time point n for an architecture trace t to the corresponding projection $\Pi_{c}(t)^\frown t'$, while *bhv2cnf* is used to map a time point n for the $\Pi_{c}(t)^\frown t'$ back to the corresponding architecture trace t.

Evaluating behavior specifications over architecture traces Now we have everything we need to formalize the evaluation operator mentioned above. It is formalized by an Isabelle function *eval*, which takes a component identifier $'id$ and a behavior trace assertion, and transforms it to a corresponding architecture trace assertion:

definition $\mathit{eval} :: {'id} \Rightarrow (nat \Rightarrow cnf) \Rightarrow (nat \Rightarrow {'cmp}) \Rightarrow nat \Rightarrow ((nat \Rightarrow {'cmp}) \Rightarrow nat \Rightarrow bool) \Rightarrow bool$
 where $\mathit{eval}\ cid\ t\ t'\ n\ \gamma \equiv (\exists i \geq n.\ \|cid\|_{t\ i}) \wedge \gamma(\mathit{lnth}\ ((\pi_{cid}(\mathit{inf\text{-}llist}\ t))@_{l}(\mathit{inf\text{-}llist}\ t')))(\mathit{the\text{-}enat}(\langle cid\ \#_{n}\ \mathit{inf\text{-}llist}\ t\rangle)) \vee$
$(\exists i.\ \|cid\|_{t\ i}) \wedge (\nexists i'.\ i' \geq n \wedge \|cid\|_{t\ i'}) \wedge \gamma(\mathit{lnth}\ ((\pi_{cid}(\mathit{inf\text{-}llist}\ t))@_{l}(\mathit{inf\text{-}llist}\ t')))(_{cid}{\downarrow}_{t}(n)) \vee$
$(\nexists i.\ \|cid\|_{t\ i}) \wedge \gamma\ (\mathit{lnth}\ ((\pi_{cid}(\mathit{inf\text{-}llist}\ t))@_{l}(\mathit{inf\text{-}llist}\ t')))\ n$

To evaluate a behavior trace assertion γ over an architecture trace t at time point n, *eval* distinguishes between three cases:

- If component *cid* is again activated in the future, γ is evaluated at the next point in time where *cid* is active in t.
- If component *cid* is not again activated in the future, but it is activated at least once in t, then γ is evaluated at the point in time given by the corresponding time mapping.
- If component *cid* is never active in t, then γ is evaluated at time point n.

4.2. Creating Pattern Theories

As mentioned at the beginning, the formalization of the model, as presented in this section, can be used to support the interactive verification of ADPs. To this end, a pattern specification (in terms of the techniques presented in Sect. 3) can be systematically transferred to a corresponding Isabelle/HOL theory. Algorithm 1 describes the mapping in more detail. In general, the transformation is done in four main steps:

1. The specified FACTum datatypes are transferred to corresponding Isabelle/HOL datatypes.
2. An Isabelle locale is created for the corresponding pattern, which imports other locales for each instantiated pattern (or locale `dynamic_component` for each type of component which does not instantiate any component type from another pattern). Ports for component types are added as locale parameters and typed by the corresponding Isabelle/HOL datatypes.
3. Specifications of component behavior are added as locale assumptions, formulated in terms of behavior trace assertions (as formalized in Sect. 4.1.4), and evaluated using the evaluation function introduced in Sect. 4.1.5.
4. Activation and connection assertions are provided as locale assumptions, formulated in terms of architecture trace assertions, formalized in Sect. 4.1.3.

The following result guarantees soundness of Alg. 1, i.e., that the algorithm indeed preserves the semantics of a FACTum specification.

Algorithm 1 Mapping a pattern specification to an Isabelle/HOL Theory.

Input: A FACTum specification of ADP
{with Datatype Spec. *DS*, Component Type Spec. *CS*, and Architecture Spec. *AS*}
Output: An Isabelle/HOL theory for the specification
1: **for all** Datatypes *dt* in *DS* **do**
2: create Isabelle/HOL datatype specification for *dt*
3: **end for**
4: create Isabelle/HOL locale for the pattern
5: **for all** Component Types *ct* in *CS* **do**
6: **if** *ct* instantiates a component type of another pattern specification *PS* **then**
7: import the corresponding locale for *PS*
{requires to import the corresponding Isabelle theory}
8: create instance of ports/parameters according to the specified port mapping
{the parameter for every port of *ct* is passed to the imported locale}
9: **else**
10: import locale "dynamic_component" of theory "Configuration_Traces"
11: **end if**
12: create instance of locale parameters *tCMP* and *active*
13: **for all** component parameters *p* of *ct* which are not instances **do**
14: create locale parameter *par* of the type corresponding to the type of *p*
15: create locale assumption "$\forall x.\ \exists c.\ par(c) = x$"
{since FACTum requires *nonempty* sets of components for each type}
16: **end for**
{instantiated parameters are already handled at line 8}
17: **for all** ports *p* which are not instances **do**
18: create locale parameter of the type corresponding to the datatype of *p*
19: **end for**
{instantiated ports are already handled at line 8}
20: **for all** behavior trace assertions *b* of *ct* **do**
21: create locale assumption for *b*
{use the operators and evaluation function presented in this chapter}
22: **end for**
23: **end for**
24: **for all** architecture trace assertions *c* of *AS* **do**
25: create locale assertion for *c*
{use the operators presented in this chapter}
26: **end for**

Theorem 4.1 (Soundness of Alg. 1). A set of architecture traces satisfies a FACTum specification iff it satisfies the theory generated from the FACTum specification by Algorithm 1.

Although a *formal* proof for this theorem is out of the scope of this text, we provide an informal argument for it in App. C. In the following, we demonstrate the algorithm by applying it to the specification of our three running examples, as presented in Sect. 3.

4.2.1. Singleton

First, we discuss the outcome generated from the specification of the Singleton pattern presented in Ex. 23.

Mapping the pattern specification. Since the specification of the Singleton did not involve the specification of data types, no Isabelle datatype specification is created. However, an Isabelle locale is created from the specification of a singleton's interface (specified in Fig. 35). The corresponding Isabelle/HOL code looks as follows:

```
locale singleton = dynamic-component cmp active
  for active :: 'id ⇒ cnf ⇒ bool (‖-‖- [0,110]60)
  and cmp :: 'id ⇒ cnf ⇒ 'cmp (σ-(-) [0,110]60) +
```

Since the Singleton pattern consists of one type of component, which does not instantiate any other component type, we instantiate locale *dynamic-component* once. To this end, we must provide two locale parameters:

- A mapping *cmp* to access a certain singleton component in a given architecture snapshot, based on its identifier.
- A predicate *active* which checks whether a singleton component, with a certain identifier, is activated in an architecture snapshot.

Moreover, the specification of the Singleton consists of two architectural constraints (specified as annotations in Fig. 35), which are transferred to corresponding locale assumptions:

```
assumes alwaysActive: ⋀k. ∃ id. ‖id‖_k
  and unique: ∃ id. ∀ k. ∀ id'. (‖id'‖_k ⟶ id = id')
```

Locale assumption *alwaysActive* is generated from the Singleton's assumption that a singleton component is always active. Locale assumption *unique* is created from the assumption that a singleton component is indeed unique.

4.2.2. Publisher-Subscriber

Next, we discuss the mapping for the specification of the Publisher-Subscriber pattern discussed in Ex. 26.

Mapping data types. Since the specification of the Publisher-Subscriber pattern indeed contains specifications for data types, we first need to create a corresponding datatype specification in Isabelle/HOL:

```
datatype 'evt subscription = sub 'evt | unsub 'evt
```

According to the datatype specification presented in Ex. 11, we create a parametric datatype *subscription*, which depends on a type parameter *'evt* to denote events for which subscribers can subscribe. Thereby, the elements of a subscription are defined to be either a subscription *sub* to an event *'evt*, or an unsubscription *unsub* for an event *'evt*.

Mapping architectural assumptions. The specification of the pattern's interfaces (as presented in Fig. 40) is again mapped to a corresponding Isabelle locale specification:

```
locale publisher-subscriber =
  pb: singleton pbactive pbcmp +
  sb: dynamic-component sbcmp sbactive
    for pbactive :: 'pid ⇒ cnf ⇒ bool
    and pbcmp :: 'pid ⇒ cnf ⇒ 'PB
    and sbactive :: 'sid ⇒ cnf ⇒ bool
    and sbcmp :: 'sid ⇒ cnf ⇒ 'SB +
```

This time, however, two interfaces are specified, which requires two locale instantiations: Since the publisher component type is specified to be an instance of the singleton component type from the Singleton pattern, a corresponding instantiation of the Singleton locale is created. The subscriber component type does not instantiate any other component type, which is why it instantiates the default locale *dynamic-component* from our framework. Note that locale instantiations are indeed transitive, which means that, implicitly, also the publisher component type instantiates locale *dynamic-component*. Thus, all the definitions of locale *dynamic-component* are available also for publisher components, although they do not directly instantiate *dynamic-component*.

In contrast to the singleton component type, which has no specified ports, publishers as well as subscribers have ports specified for their interfaces. The port types specified in Fig. 20 are mapped to corresponding locale parameters:

```
  fixes pbsb :: 'PB ⇒ ('evt set) subscription set
    and pbnt :: 'PB ⇒ ('evt × 'msg)
    and sbnt :: 'SB ⇒ ('evt × 'msg) set
    and sbsb :: 'SB ⇒ ('evt set) subscription
```

For each port, we create a locale parameter which takes a component of the corresponding component type and returns a set of messages of the corresponding port type.

Finally, the two connection assumptions, specified for the Publisher-Subscriber pattern in Fig. 40, are mapped to corresponding locale assumptions:

```
assumes conn1: ⋀k pid. pbactive pid k
    ⟹ pbsb (pbcmp pid k) = (⋃sid∈{sid. sbactive sid k}. {sbsb (sbcmp sid k)})
and conn2: ⋀t n n'' sid pid E e m.
  ⟦t ∈ arch; pbactive pid (t n); sbactive sid (t n); sub E = sbsb (sbcmp sid (t n)); n''≥ n; e ∈ E;
  ∄n' E'. n' ≥ n ∧ n' ≤ n'' ∧ sbactive sid (t n') ∧
    unsub E' = sbsb (sbcmp sid (t n')) ∧ e ∈ E';
  (e, m) = pbnt (pbcmp pid (t n'')); sbactive sid (t n'')⟧
    ⟹ pbnt (pbcmp pid (t n'')) ∈ sbnt (sbcmp sid (t n''))
```

Thereby, connection requirements between two ports are simply mapped to equality assumptions for the corresponding locale parameters. *conn1*, for example, denotes the constraint that port *sb* of a publisher component is connected to port *sb* of every active subscriber component.

4.2.3. Blackboard

Finally, we present the mapping of the Blackboard pattern as specified in Ex. 27. Again, a Blackboard pattern requires special data types, which are created first.

Mapping data types. In contrast to the Publisher-Subscriber pattern, data types for the Blackboard pattern are specified axiomatically:

```
typedecl PROB
consts sb :: (PROB × PROB) set
axiomatization where sbWF: wf sb
typedecl SOL
consts solve:: PROB ⇒ SOL
```

As required by the corresponding FACTum datatype specification (Ex. 12), the specification introduces two abstract types: problems and solutions. It then requires the existence of a well-founded relation *sb* which relates problems with corresponding subproblems. Finally, it requires the existence of a mapping *solve*, which is assumed to assign the correct solution to each problem.

Mapping architectural assumptions. Again, the pattern specification is mapped to a corresponding Isabelle/HOL locale. Similar as for the Publisher-Subscriber pattern, the pattern's interfaces are used to generate a corresponding locale header:

```
locale blackboard = publisher-subscriber bbactive bbcmp ksactive kscmp bbrp bbcs kscs ksrp
  for bbactive :: 'bid ⇒ cnf ⇒ bool (‖-‖- [0,110]60)
    and bbcmp :: 'bid ⇒ cnf ⇒ 'BB (σ-(-) [0,110]60)
    and ksactive :: 'kid ⇒ cnf ⇒ bool (‖-‖- [0,110]60)
    and kscmp :: 'kid ⇒ cnf ⇒ 'KS (σ-(-) [0,110]60)
    and bbrp :: 'BB ⇒ (PROB set) subscription set
    and bbcs :: 'BB ⇒ (PROB × SOL)
    and kscs :: 'KS ⇒ (PROB × SOL) set
    and ksrp :: 'KS ⇒ (PROB set) subscription +
  fixes bbns :: 'BB ⇒ (PROB × SOL) set
    and ksns :: 'KS ⇒ (PROB × SOL)
    and bbop :: 'BB ⇒ PROB
    and ksop :: 'KS ⇒ PROB set
```

Since the Blackboard is specified to be an instance of the Publisher-Subscriber pattern, the locale created for the Blackboard pattern instantiates the locale of the Publisher-Subscriber pattern. The instantiation requires eight parameters: The first four parameters are the usual parameters required by locale `dynamic_component` to obtain a component from an architecture snapshot, and to check activation of a component within an architecture snapshot. In addition, we must provide an additional parameter for each port available in the specification of the Publisher-Subscriber pattern. These parameters denote ports of the Blackboard pattern, which interpret the corresponding ports of the Publisher-Subscriber pattern (as specified by the port mapping provided in Fig. 41). For example, port *rp* of a blackboard corresponds to port *sb* of a publisher, port *cs* of a blackboard to port *nt* of the publisher, port *cs* of a knowledge source to port *nt* of a subscriber, and port *rp* of a knowledge source to port *sb* of a subscriber.

As a next step, we generate interface parameters and corresponding assumptions:

```
  and prob :: 'kid ⇒ PROB
assumes
  ks1: ∀ p. ∃ ks. p=prob ks — Component Parameter
```

Since knowledge sources are parameterized by problems, we must generate a corresponding locale parameter which assigns a problem to each knowledge source. Moreover, we generate an assumption *ks1*, which requires the existence of at least one knowledge source for each problem (as required by the semantics of parametric component types described in Sect. 3.2.1).

Finally, we generate additional locale assumptions, according to the activation and connection constraints described in Ex. 27:

```
  — Assertions about component activation.
  and actks:
    ⋀t n kid p. ⟦t ∈ arch; ksactive kid (t n); p=prob kid; p∈ksop (kscmp kid (t n))⟧
    ⟹ (∃ n'≥n. ksactive kid (t n') ∧ (p, solve p) = ksns (kscmp kid (t n')) ∧
    (∀ n''≥n. n''<n' ⟶ ksactive kid (t n'')))
    ∨ (∀ n'≥n. (ksactive kid (t n') ∧ (¬(p, solve p) = ksns (kscmp kid (t n')))))

  — Assertions about connections.
  and conn1: ⋀k bid. bbactive bid k
    ⟹ bbns (bbcmp bid k) = (⋃ kid∈{kid. ksactive kid k}. {ksns (kscmp kid k)})
  and conn2: ⋀k kid. ksactive kid k
    ⟹ ksop (kscmp kid k) = (⋃ bid∈{bid. bbactive bid k}. {bbop (bbcmp bid k)})
```

In contrast to the patterns discussed so far, a Blackboard involves also the specification of component types, i.e., assumptions about component behavior. Thus, one additional locale assumption is generated for every behavior assumption presented in Ex. 18 and Ex. 19, using our formalization of behavior trace assertions (as presented in Sect. 4.1.4):

```
  — Assertions about component behavior.
  and bhvbb1: ⋀t t' bId p s. ⟦t ∈ arch⟧ ⟹ pb.eval bId t t' 0
    (pb.glob (pb.ba (λbb. (p,s)∈bbns bb)
    ⟶^p (pb.evt (pb.ba (λbb. (p,s) = bbcs bb)))))
  and bhvbb2: ⋀t t' bId P q. ⟦t∈arch⟧ ⟹ pb.eval bId t t' 0
    (pb.glob (pb.ba (λbb. sub P ∈ bbrp bb ∧ q ∈ P) ⟶^p
    (pb.evt (pb.ba (λbb. q = bbop bb)))))
  and bhvbb3: ⋀t t' bId p . ⟦t∈arch⟧ ⟹ pb.eval bId t t' 0
    (pb.glob (pb.ba (λbb. p = bbop(bb)) ⟶^p
    (pb.wuntil (pb.ba (λbb. p=bbop(bb))) (pb.ba (λbb. (p,solve(p)) = bbcs(bb))))))
  and bhvks1: ⋀t t' kId p P. ⟦t∈arch; p = prob kId⟧ ⟹ sb.eval kId t t' 0
    (sb.glob ((sb.ba (λks. sub P = ksrp ks)) ∧^s
    (sb.all (λq. (sb.pred (q∈P)) ⟶^s (sb.evt (sb.ba (λks. (q,solve(q)) ∈ kscs ks)))))
    ⟶^s (sb.evt (sb.ba (λks. (p, solve p) = ksns ks)))))
  and bhvks2: ⋀t t' kId p P q. ⟦t ∈ arch;p = prob kId⟧ ⟹ sb.eval kId t t' 0
    (sb.glob (sb.ba (λks. sub P = ksrp ks ∧ q ∈ P ⟶ (q,p) ∈ sb)))
  and bhvks3: ⋀t t' kId p. ⟦t∈arch;p = prob kId⟧ ⟹ sb.eval kId t t' 0
    (sb.glob ((sb.ba (λks. p∈ksop ks)) ⟶^s (sb.evt (sb.ba (λks. (∃ P. sub P = ksrp ks))))))
  and bhvks4: ⋀t t' kId p P. ⟦t∈arch; p∈P⟧ ⟹ sb.eval kId t t' 0
    (sb.glob ((sb.ba (λks. sub P = ksrp ks)) ⟶^s
    (sb.wuntil (¬^s (∃_s P'. (sb.pred (p∈P') ∧^s (sb.ba (λks. unsub P' = ksrp ks)))))
    (sb.ba (λks. (p,solve p) ∈ kscs ks)))))
```

Remember that assumptions about component behavior are specified without considering possible activations and deactivations of a component. Thus, corresponding specifications have to be interpreted for architecture traces using the evaluation operator introduced in Sect. 4.1.5.

4.3. Verifying Architectural Design Patterns

After mapping the specification to corresponding Isabelle/HOL theories, we can verify guarantees for them. Therefore, we first specify architectural guarantees using architecture trace assertions (introduced in Sect. 3.3). Then we can map them to corresponding Isabelle/HOL theorems and verify them using Isabelle's structured proof language Isabelle/Isar [Wen07]. In the case we are dealing with pattern hierarchies, guarantees for lower

ASpec Guarantee_Singleton	**for** Singleton
flex s:	*Singleton*
rig *the-singleton*:	*Singleton*
$\exists the\text{-}singleton\colon \Box\big(\wr the\text{-}singleton\wr \wedge (\wr s\wr \longrightarrow s = the\text{-}singleton)\big)$	

Fig. 46. Architectural guarantee for the Singleton pattern.

level patterns are automatically transferred to higher level patterns where they can be used to support their verification. In the following, we demonstrate the verification phase in terms of our three running examples.

4.3.1. Singleton

Lets first discuss the verification of the Singleton pattern as specified in Ex. 23.

Guarantee. One possible guarantee for a Singleton is that there exists indeed a *unique* component of type singleton, which is always activated. It is formalized in terms of an architecture specification in Fig. 46. First, we specify two component variables for singletons: a flexible variable s and a rigid variable *the-singleton*[5]. Then, we require the existence of such a rigid singleton, which is always activated and indeed the only component of type singleton which is active at any point in time.

Mapping the guarantee. The architectural guarantee is systematically transferred to the following Isabelle/HOL theorem:

```
definition the-singleton ≡ THE id. ∀ k. ∀ id'. ‖id'‖k ⟶ id' = id

theorem ts-prop:
  fixes k::cnf
  shows ⋀id. ‖id‖k ⟹ id = the-singleton
    and ‖the-singleton‖k
```

First, Isabelle/HOL's definite description operator *THE* is used to define the *unique* singleton component. Then, a theorem is created which guarantees that a singleton is indeed unique and always activated.

Proving the theorem. A possible proof for theorem *ts-prop* in Isabelle/Isar looks as follows:

```
proof −
  { fix id
    assume a1: ‖id‖k
    have (THE id. ∀ k. ∀ id'. ‖id'‖k ⟶ id' = id) = id
    proof (rule the-equality)
      show ∀ k id'. ‖id'‖k ⟶ id' = id
      proof
        fix k show ∀ id'. ‖id'‖k ⟶ id' = id
        proof
          fix id' show ‖id'‖k ⟶ id' = id
          proof
            assume ‖id'‖k
            from unique have ∃ id. ∀ k. ∀ id'. (‖id'‖k ⟶ id = id') .
            then obtain i'' where ∀ k. ∀ id'. (‖id'‖k ⟶ i'' = id') by auto
            with ⟨‖id'‖k⟩ have id=i'' and id'=i'' using a1 by auto
            thus id' = id by simp
          qed
        qed
      qed
    next
      fix i'' show ∀ k id'. ‖id'‖k ⟶ id' = i'' ⟹ i'' = id using a1 by auto
    qed
```

[5] Recall that, flexible variables may change their value at each point in time, whereas rigid variables keep their value during the whole execution.

ASpec Publisher-Subscriber **for** Publisher-Subscriber

flex	*the-pb*:	*Publisher*
	m:	`msg`
	E:	$\wp$(`evt`)
rig	s':	*Subscriber*
	e:	`evt`

$$\Box\Big(\wr s'\wr \wedge \big(\exists E\colon\ \text{sub}\ E = s'.sb \wedge e \in E\big) \longrightarrow \Big(\big(\wr s'\wr \wedge (e,m) = \textit{the-pb}.nt \longrightarrow (e,m) = s'.nt\big)\ \mathcal{W}\ \big(\wr s'\wr \wedge (\exists E\colon\ \text{unsub}\ E = s'.sb \wedge e \in E)\big)\Big)\Big)$$

Fig. 47. Architectural guarantee for the Publisher-Subscriber pattern.

```
    hence ‖id‖_k ⟹ id = the-singleton by (simp add: the-singleton-def)
  } note g1 = this
  thus ⋀id. ‖id‖_k ⟹ id = the-singleton by simp

  from alwaysActive obtain id where ‖id‖_k by blast
  with g1 have id = the-singleton by simp
  with ‹‖id‖_k› show ‖the-singleton‖_k by simp
qed
```

Note the reference to the two assumptions *unique* (line 13) and *alwaysActive* (line 27), generated from the pattern's imposed assumptions and discussed in Sect. 4.2.1.

4.3.2. Publisher-Subscriber

Next, we discuss the verification of a possible guarantee for the Publisher-Subscriber pattern.

Results from the singleton. Since the Publisher-Subscriber pattern instantiates the Singleton pattern, results obtained for the Singleton are automatically interpreted in the context of the Publisher-Subscriber pattern. Thus, declaring the publisher to be an instance of a Singleton has two major consequences.

First, a corresponding definition of the *unique* publisher component is available:

```
abbreviation the-pb :: 'pid where
the-pb ≡ pb.the-singleton
```

Essentially, *the-pb* abbreviates definition *the-singleton* introduced in Sect. 4.3.1.

Moreover, the theorem proved for singleton components is available also for components of type publisher:

```
pb.ts-prop (1): pbactive id k ⟹ id = the-pb
pb.ts-prop (2): pbactive the-pb k
```

Guarantee. A possible guarantee for our version of the Publisher-Subscriber pattern is specified in Fig. 47. It guarantees that a subscriber component indeed receives all the messages for which it is subscribed.

Mapping the guarantee. Similar as for the Singleton pattern, the architectural guarantee for Publisher-Subscriber architectures can be mapped to a corresponding Isabelle/HOL theorem:

```
theorem msgDelivery:
  fixes t n n'' sid E e m
  assumes t ∈ arch
    and sbactive sid (t n)
    and sub E = sbsb (sbcmp sid (t n))
    and n'' ≥ n
```

```
  and ∄n' E'. n' ≥ n ∧ n' ≤ n'' ∧ sbactive sid (t n') ∧ unsub E' = sbsb(sbcmp sid (t n')) ∧ e ∈ E'
  and e ∈ E
  and (e,m) = pbnt (pbcmp the-pb (t n''))
  and sbactive sid (t n'')
shows (e,m) ∈ sbnt (sbcmp sid (t n''))
```

Proving the theorem. The proof for theorem *msgDelivery* in Isabelle is a simple one-liner:

```
using conn1[OF pb.ts-prop(2)] .
```

It follows directly from the assumptions generated for the pattern and the guarantee inherited from the Singleton.

4.3.3. Blackboard

Finally, we discuss the verification of a guarantee for the Blackboard pattern.

Results from the Publisher-Subscriber pattern. Since a Blackboard instantiates the Publisher-Subscriber pattern (and therefore implicitly also the Singleton pattern), we get all the properties verified for these patterns also for the Blackboard pattern, for free. First, we get all the results for the Singleton pattern:

```
abbreviation the-bb ≡ the-pb
pb.ts-prop (1): ‖id‖_k ⟹ id = the-bb
pb.ts-prop (2): ‖the-bb‖_k
```

Similar as for the Publisher-Subscriber, we first introduce an abbreviation *the-bb* to denote the *unique* component of type blackboard and then we get a corresponding lemma, about uniqueness of a blackboard component, from the guarantee of the Singleton pattern.

In addition, we get results from the Publisher-Subscriber pattern:

```
msgDelivery:
  ⟦t ∈ arch;
  ‖sid‖_t n;
  sub E = ksrp (σ_sid t n);
  n ≤ n'';
  ∄n' E'. n ≤ n' ∧ n' ≤ n'' ∧ ‖sid‖_t n' ∧ unsub E' = ksrp (σ_sid t n') ∧ e ∈ E';
  e ∈ E;
  (e, m) = bbcs (σ_the-bb t n'');
  ‖sid‖_t n''⟧
    ⟹ (e, m) ∈ kscs (σ_sid t n'')
```

Basically, the results resemble the property verified for Publisher-Subscriber patterns (discussed in the last section) with activation and selection parameters from knowledge sources and blackboards, respectively.

Guarantee. Figure 48 provides the specification of a possible guarantee for Blackboard architectures: If for every open subproblem, a knowledge source able to solve this problem is eventually activated (Eq. (27)), then, the architecture will eventually solve a given problem (Eq. (28)), even if no single knowledge source is able to solve the problem on its own). Note that the specification uses the concept of parameterized variables for knowledge sources. Thus, given a problem p, variable $ks_{\langle p \rangle}$ denotes a variable for a knowledge source component which is indeed able to solve problem p.

Mapping the guarantee. As for the other examples, we finally generate a theorem according to the guarantee presented in Sec. 4.3.3. To foster readability, we first use Isabelle/HOL's indefinite description operator *SOME* to introduce an additional definition to denote a knowledge source with a certain property:

```
definition sKs:: PROB ⇒ 'kid where
  sKs p ≡ (SOME kid. p = prob kid)
```

Then, we can use this definition to create a corresponding Isabelle/HOL theorem:

```
theorem pSolved:
  fixes t and t'::nat ⇒'BB and t''::nat ⇒'KS
```

ASpec Guarantee_Blackboard		**for** Blackboard
flex	*the-bb*:	*BB*
	ks:	*KS*
	P:	$\wp$(PROB)
rig	*p*:	PROB

$$\Big(\Box\big(\forall p = \textit{the-bb.op}\colon \Diamond\wr ks_{\langle p\rangle}\wr\big)\Big) \longrightarrow \qquad (27)$$

$$\Box\Big(\forall P\colon \big(\texttt{sub}\ P \in \textit{the-bb.rp} \longrightarrow \forall p \in P\colon \Diamond\big((p, solve(p)) \in \textit{the-bb.cs}\big)\Big) \qquad (28)$$

Fig. 48. Architectural guarantee for the Blackboard pattern.

```
assumes t∈arch and
  ∀ n. (∃ n′≥n. ksactive (sKs (bbop(bbcmp the-bb (t n)))) (t n′))
shows
  ∀ n. (∀ P. (sub P ∈ bbrp(bbcmp the-bb (t n))
    ⟶ (∀ p ∈ P. (∃ m≥n. (p,solve(p)) = bbcs (bbcmp the-bb (t m))))))
```

Proving the theorem. To prove theorem *pSolved*, we first prove a corresponding lemma:

```
lemma pSolved-Ind:
  fixes t and t′::nat ⇒′BB and p and t′′::nat ⇒′KS
  assumes t∈arch and
    ∀ n. (∃ n′≥n. ksactive (sKs (bbop(bbcmp the-bb (t n)))) (t n′))
  shows
    ∀ n. (∃ P. sub P ∈ bbrp(bbcmp the-bb (t n)) ∧ p ∈ P) ⟶
      (∃ m≥n. (p,solve(p)) = bbcs (bbcmp the-bb (t m)))
```

The lemma can be proved by well-founded induction over the subproblem relation sb, since *sb* was declared to be well-founded in Sec. 4.2.3. The final theorem is now a direct consequence of this lemma and can be proved in a single line:

```
using assms pSolved-Ind by blast
```

4.4. Summary

In this section, we described our approach for the interactive verification of ADPs, based on interactive theorem proving. To this end, we first presented our formalization of the model of dynamic architectures (from Sect. 2) in Isabelle/HOL. The corresponding theory is based on the theory of coinductive lists and contains formalizations of all the concepts introduced for the model in Sect. 2. The formalization also provides an interface to the concepts of the model in terms of an Isabelle locale `dynamic_component`, which can be instantiated to obtain specification operators for component types.

Next, we presented an algorithm to map a FACTum specification (as presented in Sect. 3) to a corresponding Isabelle/HOL theory. The resulting theory imports the theory of our model, contains corresponding datatype specifications, and a locale for the pattern specification. The locale instantiates `dynamic_component` for each specified type of component and contains assumptions derived from the corresponding specification of component types and architectural constraints.

Finally, we demonstrate the approach in terms of our three running examples: We demonstrated the Isabelle/HOL specification generated for each pattern, formalized a corresponding guarantee in FACTum, and proved it using Isabelle's structured proof language Isabelle/Isar. Since our approach supports hierarchical pattern specifications, guarantees proved for a pattern are automatically transferred to higher level patterns, where they can be used to support the verification of these patterns. Thus, verification of the Publisher-Subscriber was supported by the guarantee inherited from the Singleton pattern and verification of the Blackboard pattern used the results inherited from the Publisher-Subscriber pattern.

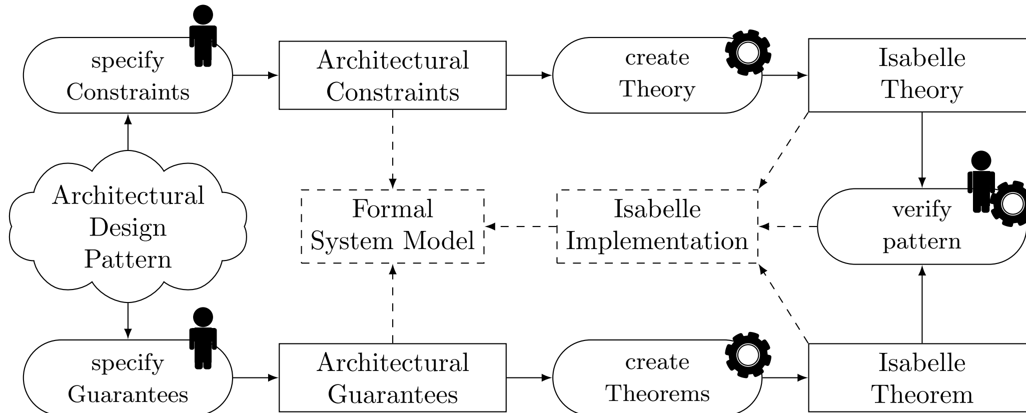


Fig. 49. The FACTum methodology for interactive pattern verification.

5. Methodology and Evaluation

So far, we have introduced a formal model for dynamic architectures (Sect. 2), techniques to specify ADPs over this model (Sect. 3), and a framework to support the verification of such specifications, based on the interactive theorem prover Isabelle/HOL. In the following section, we elaborate on these results and integrate them into an overall methodology for the interactive verification of ADPs.

Figure 49 depicts the main activities (rounded rectangles) and artifacts (rectangles) of the FACTum methodology for interactive pattern verification. It consists of three main phases:

1. Pattern specification
2. Theory creation
3. Pattern verification

In the *pattern specification* phase, an ADP is specified by formalizing the corresponding architectural constraints and architectural guarantees. As indicated by the stick figures, specification is a manual activity. The specification is based on the model presented in Sect. 2 and supported by the techniques described in Sect. 3. The outcome of this first phase is a formal specification of a pattern's architectural constraints and architectural guarantees. The formal specification of the pattern is then used in the *theory creation* phase to create corresponding Isabelle/HOL theories. The theories are built on top of our formalization of architecture traces, presented in the last section, and their generation can be automated by the algorithm presented in Sect. 4.2. In the *pattern verification* phase, the pattern is verified by proving the theorem (obtained from the pattern's guarantee) from the pattern's specification. As indicated by the figure, this is a semi-automated step: A user first develops a corresponding proof in Isabelle's structured proof language Isar. Then, the logical reasoners of Isabelle ensure soundness of each of the steps performed in the proof.

To support a user in the specification and theory creation phase, FACTum is accompanied by a tool called FACTum Studio. To this end, we implemented the specification techniques presented in Sect. 3 in Eclipse/EMF. In addition, we implemented the algorithm presented in Sect. 4.2 to automatically generate Isabelle/HOL theories from specifications developed in FACTum Studio. We evaluated FACTum by means of three example cases and one larger case study. In the following, we first present FACTum Studio in more detail. Then, we present the results obtained from our evaluation of FACTum.

5.1. FACTum Studio

FACTum Studio is an Eclipse/EMF-based application that implements the approach presented so far to support the specification and verification of architectural design patterns. At the time of writing, the tool's main features range over the modeling of design patterns to transforming these models into Isabelle/HOL code [MG18]. In the following, we briefly discuss its main features and provide an overview of its architecture.

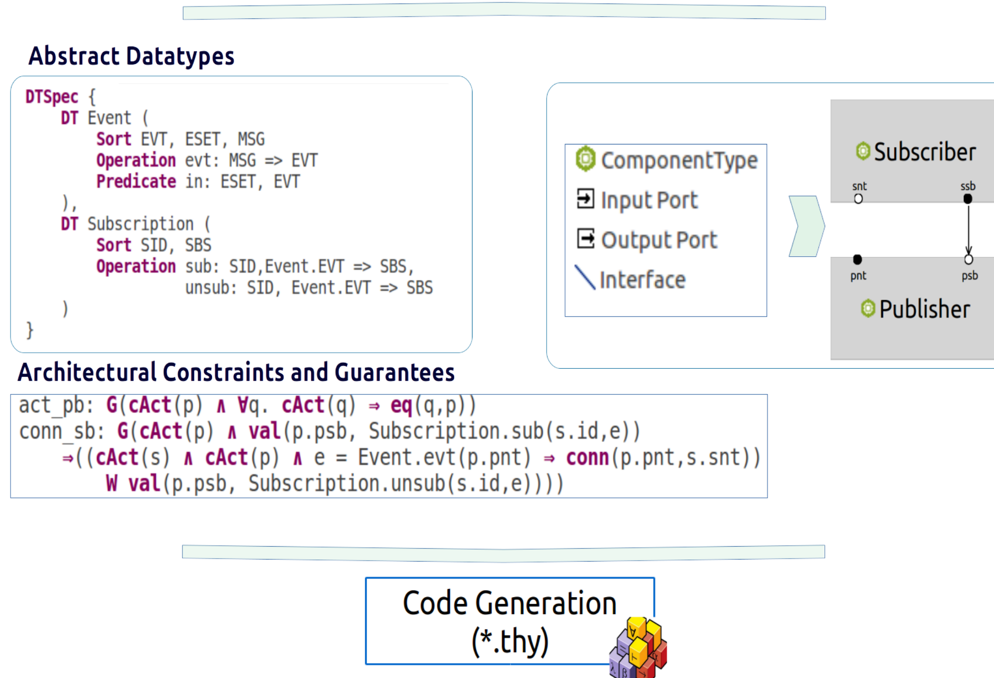


Fig. 50. FACTum Studio Features.

5.1.1. Overview of Main Features

Figure 50 illustrates the tools main features by means of the Publisher-Subscriber example:

- Textual specification of abstract datatypes (as discussed in Sect. 3.1).
- Graphical modeling of interfaces for component types (as discussed in Sect. 3.2.1) and textual specification of corresponding behavioral constraints (as discussed in Sect. 3.2.2).
- Textual specification of architectural constraints and architectural guarantees (as discussed in Sect. 3.3).
- Generation of Isabelle/HOL theories from specifications (according to Alg. 1).

To support the user in the development of correct specifications, FACTum Studio provides support for various types of static checks: Whenever datatype functions or predicates are used for the specification of component types, architectural constraints or architectural guarantees, they are checked for type-consistency according to their signature declaration. Terms are typed by sorts from corresponding datatype specifications and they are checked for type-compatibility when used in the specification of component types, architectural constraints, or architectural guarantees. Behavioral specifications for a certain component type must only use ports of that type. Ports which are required to be connected must be typed by the same sort.

5.1.2. FACTum *Studio Architecture*

FACTum Studio is implemented based on the Eclipse Modeling Framework (EMF), particularly the Obeo Designer Community edition [TO17]. The development of the tool utilizes frameworks and languages in the Eclipse/EMF ecosystem, which include Xtext, Ecore, Xtend and Sirius [Bet16, TO17].

Figure 51 illustrates the architecture of FACTum Studio. The tool has two workspaces. The first part is where the domain language is modeled and defined, which is the FACTum pattern metamodel. It is the development workspace where the grammar, constraints, validations, and the code generation templates are developed. Additionally, the workspace contains standard EMF editors that support the development of the domain metamodel. The second part, the FACTum pattern instance, is the user workspace or the runtime of the domain language, where patterns are modeled based on the metamodel. The user workspace provides

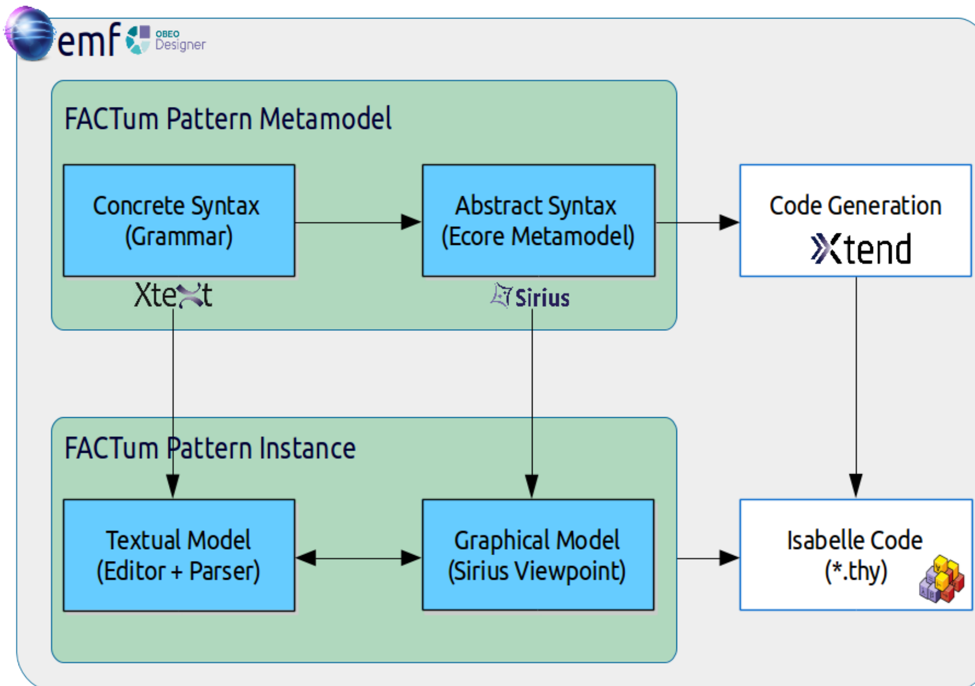


Fig. 51. FACTUM Studio Architecture.

textual and graphical editors for the specification of ADPs as discussed in Sect. 3. It also enables code generation from the specified model.

FACTUM grammar. The grammar is the FACTUM language definition, which contains all the production rules describing the concrete syntax. It includes the concrete syntax of abstract datatypes, component types (including their interfaces and behavior), architectural constraints, and architectural guarantees. All the other FACTUM language infrastructure elements are mainly generated from the FACTUM grammar. It includes the Ecore model, the parser and text editor, and other additional common language infrastructure code.

The FACTUM grammar implementation is developed based on Xtext. Xtext is a language engineering framework within the Eclipse Modeling Framework [Bet16]. It provides an infrastructure for parsing, linking, type checking, and an editor which works smoothly with the Eclipse-based IDE. The FACTUM grammar file, Pattern.xtext, is available in the project's source code repository in Github [GM18].

Ecore metamodel. The Ecore Metamodel is the central or core part of EMF-based languages. It describes domain models in the form of classes, attributes, and their relationships [TO17]. Additionally, it provides a structured data model to support the generation of abstract syntax trees and a set of Java classes based on the grammar.

In FACTUM, the Ecore metamodel is generated automatically from the grammar as an Xtext language artifact and named (Pattern.ecore) [GM18]. Additional language elements such as referencing, validation, constraint checks, and type checking are implemented based on the Ecore model. The implementation of constraint checks, validation, and other language support is developed using Xtend [Bet16].

Code generation. Based on the parsed Ecore model, models can be interpreted and transformed into other language codes or models. Code generation in EMF is implemented with the Xtend template engine. After the desired transformation mapping is developed, EMF automatically integrates the code generator into the runtime workspace.

Similarly, the FACTUM code generator template implementation is developed, based on Xtend. It gen-

erates Isabelle/HOL theories, based on a specified pattern. The FACTum code generator template files are available in the project source code repository in Github [GM18].

Textual model. The FACTum textual editor provides the support to specify models of ADPs in a textual format. Starting with a definition of data types, it enables specification of all other elements of patterns, such as component types, architectural constraints, and architectural guarantees. The text editor is part of Xtext generated language artifacts. The FACTum Textual model example files are available in the project code repository [GM18].

Graphical model. Similarly to the text editor, the FACTum graphical editor enables the specification of pattern elements but graphically. It provides a list of graphical items in a palette, where a user can drag and create pattern elements. Currently, the graphical editor does not enable specification of behavior and constraints, which must be defined in the textual editor. The graphical editor is developed based on Sirius. It is a graphical modeling and visualization tool based on EMF. Sirius graphical editors are developed by configuring viewpoint specifications on Sirius.

Changes to the graphical model are automatically reflected in the textual model and vice versa. The synchronization is triggered, once the model is saved. The FACTum Graphical model example files are available in the project code repository [GM18].

5.2. Evaluation

In order to evaluate our approach, we applied it to specify and verify four architectural design patterns: versions of the Singleton, the Publisher-Subscriber, and the Blackboard pattern, as well as a pattern for Blockchain architectures. The corresponding Isabelle theories are available in the entry `Architectural_Design_Patterns` [Mar18b] from the archive of formal proofs. Figure 52 depicts an overview of the corresponding verification efforts: Verification of the Singleton pattern (ST) consists of 180 lines of Isabelle/HOL proof code, distributed amongst nine different propositions. The Publisher-Subscriber pattern (PS), on the other hand, was verified in two propositions amounting up to 90 lines of proof code. With 547 lines of proof code, verification of the Blackboard pattern (BB) is more extensive than the verification of the Singleton as well as the Publisher-Subscriber pattern. Its verification is distributed amongst four different propositions. The most effort was required for the verification of our pattern for Blockchain architectures (BC). Its verification required $2,623$ lines of proof code distributed amongst 72 different propositions.

Figure 52 shows that the verification of the Blockchain pattern was the most extensive one. However, one needs to be careful when interpreting the data depicted in the figure, since the verification of the other patterns was done in a hierarchical manner. Thus, verification results from the Singleton pattern were reused for the verification of the Publisher-Subscriber pattern, which results in a lower effort for the verification of the latter. Similarly, all the results from the Publisher-Subscriber pattern (and in turn those for the Singleton pattern) were reused for the verification of the Blackboard pattern.

In the following, we are going to describe the verification of each pattern in more detail. First, we discuss the verification of the Singleton, the Publisher-Subscriber, and the Blackboard pattern. Then, we conclude the section with a discussion of the verification of the Blockchain pattern.

5.2.1. An Initial Pattern Hierarchy

As already mentioned above, verification of the Singleton pattern, the Publisher-Subscriber pattern, and the Blackboard pattern build on top of each other and form what we call a pattern hierarchy. Figure 53 shows the verification effort for each of the patterns in more detail: It shows the number of lines of proof code for each proposition of each of the three patterns. In the following, we are going to discuss this hierarchy in more detail, starting with the Singleton pattern, we proceed with the Publisher-Subscriber pattern, and conclude our discussion with the Blackboard pattern.

Singleton. The leftmost graph in Fig. 53 depicts the effort for the verification of the Singleton pattern. Essentially, we have two classes of verification results for this pattern:

- A key result for the pattern is formalized by property `ts_prop`, which guarantees that, in our version of the Singleton, a singleton component is *unique* and always *active*.

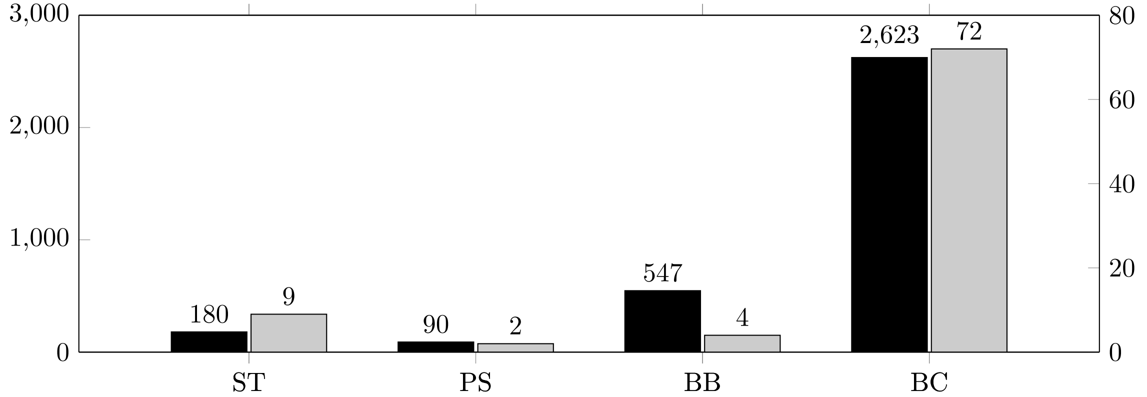


Fig. 52. Lines of proof code (black) and number of propositions (gray) for the verification of four ADPs.

- The second class of results leverages property `ts_prop` to provide rules to reason about the behavior of singleton components. Remember that, in general, reasoning about the behavior of components requires to consider activation constraints for that type of component. However, since a singleton component is always active and unique, reasoning about its behavior can be done without considering activation specifications, at all.

Publisher-Subscriber. In our version of the Publisher-Subscriber pattern, we modeled a publisher component as an instance of a singleton. Thus, as already mentioned above, all the verification results for singleton components from the Singleton pattern are available for publisher components in the Publisher-Subscriber pattern. Hence, we get special rules to reason about the behavior of publisher components which we use for the verification of two additional properties for a Publisher-Subscriber architecture:

- Property `msgDelivery` provides a characteristic property for such architectures which ensures that a subscriber component indeed receives all the messages associated with events for which it is subscribed.
- Property `conn1A` provides a more technical result to support the reasoning about Publisher-Subscriber architectures. It leverages the fact that a publisher is actually a singleton to provide an alternative version of the basic rule to reason about connected components.

As can be observed from Fig. 53, the proofs for both properties are simple one-liners. Note, however, that this is only due to the fact that the proofs are based on the results obtained from the Singleton pattern.

Blackboard. We modeled the Blackboard pattern as a version of the Publisher-Subscriber pattern in which a blackboard takes the role of a publisher, and the knowledge sources correspond to subscriber components. Thus, again, all the results from the Publisher-Subscriber pattern are available to support the verification of the Blackboard pattern. The additional constraints added by the Blackboard pattern can then be used to derive some additional guarantees of which one is of particular interest: Property `pSolved` guarantees that a Blackboard architecture is able to solve a given problem, given that for each open sub-problem, there exists a knowledge source which is able to address it. To proof this property, we first proved a more general result `pSolved_Ind` by induction. The proof of it consisted of 391 lines of Isabelle/HOL code. The final property then follows directly from this lemma.

5.2.2. A Pattern for Blockchain Architectures

To evaluate the approach on a larger case study, we formalized a pattern for Blockchain architectures based on the proof-of-work consensus algorithm [Nak08]. We then verify a characteristic property for such architectures: *persistence* of blockchain entries [KRDO17]. To deal with the probabilistic nature of Blockchain, we formalized an assumption about relative mining frequencies of honest and dishonest nodes, which allowed us to verify persistence of entries in a non-probabilistic way. Of course, the validity of that assumption to be true within a certain environment is probabilistic, but if it can be proven with a certain probability, then, our results can be used to conclude that entries are persistent with that probability. While a detailed discussion

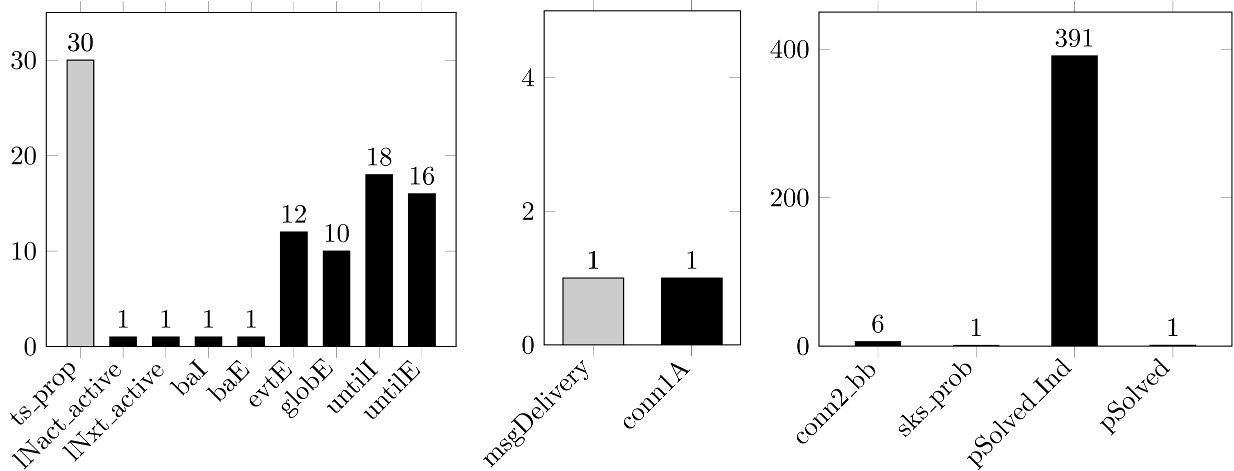


Fig. 53. Propositions for the Singleton pattern, the Publisher-Subscriber pattern, and the Blackboard pattern.

of the blockchain pattern is beyond the scope of this paper, in the following, we provide an overview of the required effort to verify the pattern.

The verification is split into three different Isabelle/HOL theories available at [Mar18b]:

- A theory `Auxiliary` which contains some auxiliary results, such as custom induction rules.
- A theory `RF_LTL` which contains a calculus for Blockchain architectures based on counting LTL [LMP10].
- A theory `Blockchain` which is the main theory containing the actual formalization of the pattern.

Figure 54 depicts the effort of the corresponding verification in terms of proof code for each proposition. The key property is formalized as theorem *blockchain-save* (highlighted with gray color in the figure). Its proof is by induction and consists of roughly 300 lines of Isabelle/HOL proof code. It required to introduce two auxiliary concepts:

- a set `his` containing a blockchain's history, i.e., its state during certain points in time and
- a function `devBC`, representing a blockchain's development.

The main remaining propositions are then concerned with these two concepts:

- Lemma `his_determ_ext` shows that the history of a blockchain is deterministic, i.e., that it has a unique state at each point in time.
- Lemma `devExt_devop` proves a basic property for blockchain developments, i.a., that it can only grow by one through a mining process.
- Lemma `devExt` shows that the development of a blockchain (which is defined using its history), is indeed a well-defined function.

6. Related Work

As mentioned in the introduction, architecture description languages (ADLs) have been an active area of research and many approaches emerged to support the formal specification of architectures. Famous examples are Weaves [GR91], Rapide [LKA+95], Wright [All97], AADL [FLV06], Acme [GMW00], xADL [DVdHT01], and Autofocus [HF10]. Over the last years, specification and verification of dynamic aspects were of particular interest. Table 2 provides an overview of some representative examples in this area. For each of them, we list the underlying formalism as well as its support for dynamic aspects. To this end, we distinguish between *Combined* and *Separate* approaches: While the former combine the specification of behavior with the specification of architectural aspects, the latter strictly separate these two.

Similar to most of the approaches shown in Tab. 2, we also separate the specification of behavioral aspects from that of structural aspects. The difference comes, however, in the verification: While most of

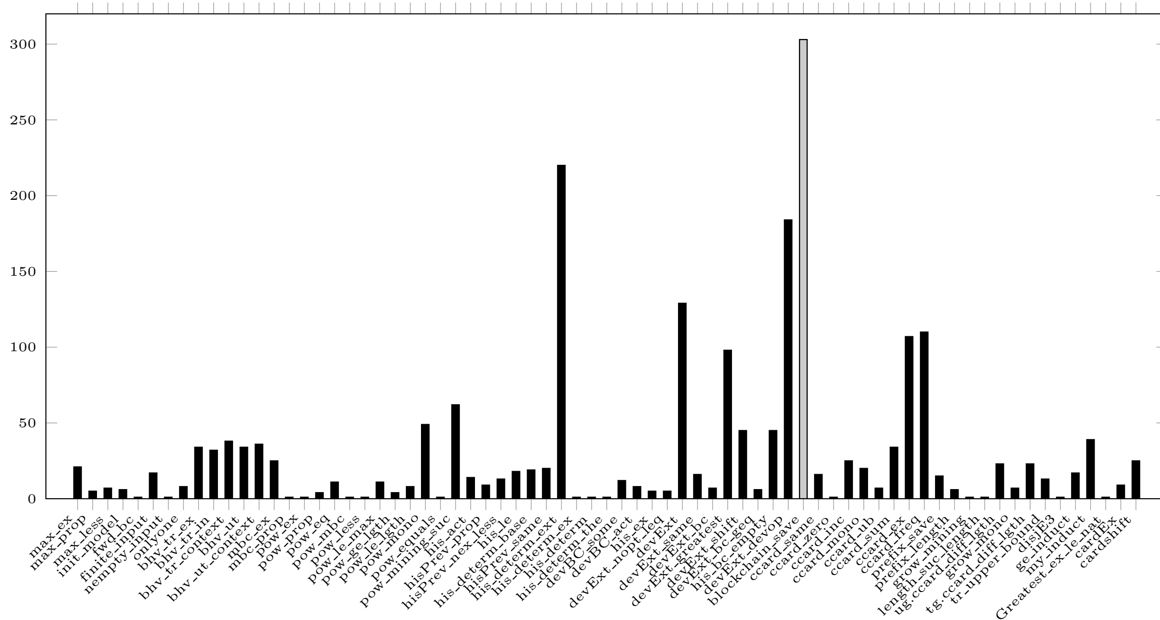


Fig. 54. Propositions for the Blockchain pattern.

approach	dynamics	specification
Darwin [MK96]	S & C	Π-Calculus [Mil99]
Wright [All97, ADG98]	S	CSP[Hoa78]
COMMUNITY [WLF01, WF02]	S	Unity [Cha89]/SM
Aguirre and Maibaum [AM02b, AM02a]	S	TL [MP92]
Π-ADL [Oqu04]	S	Π-Calculus [Mil99]
Reo [Arb04, BSAR06, KMLA11]	S	circuits
Castro et. al [CAPM10]	C	Category Theory
Canal et al. [CCS12]	S	LTS
Archery [SBR12, SMB15]	S	ACP [BK86]

Table 2. Overview of dynamic ADLs and Coordination Languages.

these approaches focus on operational specifications and automatic verification techniques, with our work we aim towards axiomatic specifications and interactive theorem proving.

6.1. Axiomatic approaches

Even though they were not invented for the purpose of pattern verification, there exist some approaches which focus on the axiomatic specification of architectures, in general. One of the first attempts in this direction is done by Bergner [Ber96]. The author proposes an approach to specify component networks and verify whether a given (runtime) component network satisfies its specification. The approach is implemented in Spectrum [BFGea93], a functional programming language which allows for axiomatic specifications of functions. Another approach comes from Fensel and Schnogge [FS97], which apply the KIV interactive theorem prover [Rei95] to verify concrete architectures in the area of knowledge-based systems. Another example is Spichkova [Spi07], which provides a mapping from a FOCUS [BS01] specification to a corresponding Isabelle/HOL [NPW02] theory. More recently, some attempts were made to apply interactive theorem proving to the verification of architectural connectors. Li and Sun [LS13], for example, apply the Coq proof assistant [BC13] to verify connectors specified in Reo [Arb04]. These approaches, both, apply interactive theorem proving to verify architectures.

While also these approaches indeed support axiomatic specifications and verification of architectures, there are two major differences to our work.

6.1.1. Scope of Application

The first difference lies in the scope of the application: The approaches discussed so far apply axiomatic verification at the level of concrete architectures which might be too expensive, in general. Thus, we argue, that application of axiomatic verification should be restricted to architecture patterns, rather than concrete architectures. Thus, the expenses would pay off since each result at the level of pattern applies for each concrete architecture implementing the pattern. Just think about how many patterns employ a Singleton or Publisher-Subscriber pattern.

6.1.2. Dynamic Aspects

Another difference lies in the expressiveness of the specification languages: The above approaches mainly focus on the specification of static architectures. However, as shown at the beginning, some commonly used patterns require also the specification of dynamic aspects, such as:

Component Activation Some patterns, such as the Singleton pattern or the Blackboard pattern introduced in Sect. 1, require to specify activation and deactivation of components.

Reconfiguration Other patterns, such as the Publisher-Subscriber pattern or the Blackboard pattern, require means to specify architecture reconfiguration, i.e., means to specify activation and deactivation of connections between component ports.

There are two exceptions to this which support axiomatic specifications of even dynamic architectures. They are closely related to our approach and thus deserve a detailed analysis.

6.2. Componentware

One example which uses a model similar to ours to formalize UML models is Componentware [Rau01]. Here, the author provides means to specify architectural constraints in an UML-like notation [RJB04]. There are, however, some differences to our specification approach which makes the specification of patterns difficult:

- The main restriction is probably the use of OCL for the specification of the behavior of components. As our examples later on show, specifying component types involves the specification of temporal aspects which is not supported by OCL and consequently not possible in their approach.
- Another restriction is the limited possibility for analysis of specifications. The approach does not provide any calculus to analyze an axiomatic specification.

Nevertheless, the approach provides many interesting insights into axiomatic specification of dynamic architectures and indeed the underlying model of dynamic architectures is similar to the model used in the approach presented with this paper.

6.3. Community

Another, closely related approach is the one of Aguirre and Maibaum [AM02b, AM02a]. The approach builds on top of CommUnity [FM97] and provides many interesting ideas found in our approach as well:

- It allows for the specification of abstract data types used by the components.
- It allows for the specification of classes which are similar to our notion of component types.
- Instance of classes as well as reconfigurations can then be specified using so-called subsystems which are similar to our notion of architecture constraint specification.

There are, however, some subtle differences to our approach which limits its application for the specification of patterns:

- Instantiation of components as well as architecture reconfiguration must be explicitly triggered from outside. However, as shown later on, for some patterns there is no such well-defined trigger, i.e., the trigger may change in different implementations of the pattern.
- Their approach doesn't support the notion of parametric interfaces which turn out to be useful when it comes to the specification of patterns.
- The approach does not support hierarchical specifications which are very important when it comes to the specification of patterns since they are usually specified on top of each other.
- The approach is based on an action-synchronous model of systems. Some patterns are, however, better described using a time-synchronous model of communication.

Another key constraint of this approach is the lack of analysis methods to reason about such specifications.

7. Conclusion

This paper introduced FACTUM, a methodology for the axiomatic specification and verification of architectural design patterns (ADPs). To conclude, we first summarize the major results presented in this paper and discuss possible implications thereof. Then, we describe our overall research agenda and point to future work which is needed to achieve our vision.

7.1. Summary

With this paper we present a model for (potentially dynamic) architectures and techniques to specify ADPs over this model. Then, we introduced an Isabelle/HOL-based framework for the interactive verification of architectures and provided an algorithm to map a pattern specification to a corresponding Isabelle/HOL theory. To evaluate the approach, we implemented it in terms of an Eclipse/EMF application and applied it for the verification of four different ADPs: the Singleton, the Publisher-Subscriber, the Blackboard pattern, and a pattern for Blockchain architectures.

7.1.1. A Model of Dynamic Architectures

Since patterns exist for static as well as for dynamic architectures, our approach is based on a model of dynamic architectures, which is described in detail in Sect. 2. Our model is a dynamic version of Broy's FOCUS model [BS01] and consists of the following main concepts:

- messages and ports (typed with sets of messages),
- interfaces consisting of input and output ports,
- a set of component types which consist of an interface, component parameters valuated with messages, and associated behavior in terms of a causal set of behavior traces, i.e., streams of snapshots of a component during execution,
- an architecture specification consisting of a set of architecture traces, i.e., streams of snapshots of an architecture during execution,
- a projection operator, which extracts the behavior of a single component out of a given architecture trace, and
- a composition operator which combines a set of component types with a given architecture specification.

7.1.2. Basic Specification Techniques

Based on the model presented in Sect. 2, we describe basic techniques for the axiomatic specification of ADPs in Sect. 3. Such a specification consist of three parts: an interface specification, a component type specification, and a specification of architectural constraints.

Interface specification. An interface specification consists of a specification of the abstract data types used in a pattern, a set of port identifiers typed by these data types, and a set of interfaces over these ports.

Data types are specified using traditional, algebraic specification techniques [Bro96], and interfaces can be specified using a graphical specification language called *architecture diagrams*.

To support the specification of related types of components (which is often required for the specification of ADPs), we also provide a notion of *parameterized* component types. Therefore, interfaces may contain so-called *interface parameters* which are typed by the abstract datatypes introduced for the pattern. Their semantics requires that at least one component exists for each valuation of interface parameters, which allows to introduce the notion of parameterized component variables. Such variables are guaranteed to be interpreted only by components with corresponding parameter values and thus support the specification of component types and architectural assumptions.

Component type specification. A component type specification consists of a set of axioms for each interface to specify assertions about the behavior of components of a certain type. To specify these axioms, we introduce the notion of *behavior trace assertion*, a type of linear temporal logic with component ports as free variables.

Specifying architectural constraints. Architectural constraints are formulated over all interfaces to specify assumptions about the activation/deactivation of components and their connections. To specify these axioms, we introduce the notion of *architecture trace assertions*, which are again a type of linear temporal logic with special predicates to denote component activation/deactivation and connections between the ports of components.

Advanced specification techniques. To facilitate the specification of certain activation and connection constraints, we propose various types of *annotations* for *architecture diagrams*. In general, annotations are graphical synonyms for corresponding architectural assertions and their semantics is given by mapping them to corresponding architecture trace assertions.

7.1.3. Hierarchical Specifications

A pattern specification may reuse other pattern specifications by instantiating the corresponding component types. Pattern instantiations are expressed by annotating the interfaces of architecture diagrams by so-called *port mappings*, i.e., mappings which relate the ports of instantiated components with the ports of the corresponding instantiating component.

Hierarchical specifications can be used, for example, to address the specification (and verification) of pattern variants. To this end, a variant of an existing pattern can be easily specified by adding additional constraints about behavior, activation, and/or connections. For example, a basic variant of the Model-View-Controller pattern [BMR+96] could be specified without direct coupling between model and views. Then, one can extend this basic version with a constraint requiring the existence of connections between model and view components.

7.1.4. Interactive Pattern Verification in Isabelle/HOL

Section 4 describes a framework for the interactive verification of ADPs specified using the techniques presented in Sect. 3. To this end, we first present a formalization of the model introduced in Sect. 2 in Isabelle/HOL. Then we describe an algorithm to map a given pattern specification to a corresponding Isabelle/HOL theory, preserving the specification's semantics (as described in Sect. 3). To consider hierarchical specifications, a pattern specification is mapped to a corresponding Isabelle/HOL locale [Bal04], which is Isabelle's way of dealing with parameterized theories. Thus, verification results for lower level patterns are automatically transferred to higher level ones where they can be used for the verification of those patterns.

7.1.5. The FACTum Methodology

Section 5 combines all the results from the previous sections into an overall methodology for the axiomatic specification and interactive verification of ADPs. The methodology consists of three phases:

Pattern Specification First, a pattern is specified using the techniques presented in Sect. 3.

Theory Creation Then, a corresponding Isabelle/HOL theory is generated using the algorithm presented in Sect. 4.

Pattern verification Finally, a pattern is verified using Isabelle's structured proof language Isar [Wen07].

To support an architect in the specification of ADPs, FACTUM comes with tool support in terms of a corresponding Eclipse/EMF implementation. FACTUM Studio supports the graphical modeling of architecture diagrams (as described in Sect. 3), which can then be enriched by corresponding textual specifications. To support these textual specifications, the tool also provides rigorous type checking mechanisms for the specification of datatypes, component types, and architectural assumptions (as described in Sect. 3). Finally, the tool implements the algorithm presented in Sect. 4 to generate corresponding Isabelle/HOL theories.

7.1.6. Running Examples & Case Study

Our approach is demonstrated in terms of three running examples and evaluated by means of a larger case study from the domain of blockchain architectures.

Running examples. We demonstrated our approach by means of three running examples: the Singleton, the Publisher-Subscriber, and the Blackboard pattern. For each pattern, we provided a formal specification of the pattern's assumptions and corresponding guarantees. Then, we verified each of them in Isabelle/HOL which resulted in a verification effort of around $3,500$ lines of Isabelle/HOL proof code. To demonstrate hierarchical specification and verification, the publisher component is modeled as an instance of the Singleton and the Blackboard pattern is specified as an instance of the Publisher-Subscriber pattern.

Case study: Verified blockchain architectures. To evaluate the approach on a larger case study, we applied it to verify a pattern for Blockchain architectures. To this end, we first applied the specification techniques from Sect. 3 to formalize the patterns assumptions as well as an important guarantee for Blockchain architectures: persistence of blockchain entries. We then map the specification to a corresponding Isabelle/HOL theory and the guarantee to a corresponding Isabelle/HOL theorem (using the algorithm presented in Sect. 4) and verify the guarantee by proving the theorem. Verification of this pattern resulted in roughly $3,000$ lines of Isabelle/HOL proof code.

7.2. Implications

The methodology presented in this paper can be used to formally investigate ADPs. Thereby, we address both problems with pattern specifications identified in Sect. 1.

7.2.1. Problem 1: Missing Constraints

Verifying an ADP may reveal constraints assumed by the pattern which are important to meet its guarantee, but which are not mentioned in any specification of the pattern so far. While the major part of such missing constraints is usually concerned with details of an architecture, some of them can also be of more fundamental nature. For example, in this paper, we discover around 16 assumptions for different ADPs. While many of them are concerned with details about an architecture, two of them may be considered fundamental: The first one is assumed by Blackboard architectures and requires problems to be ordered by a subproblem relation which is required to be *well-founded*. This is a fundamental constraint which needs to be ensured before applying the pattern. Otherwise, the corresponding architecture will not be able to solve certain problems and the pattern would not fulfill its purpose. A second fundamental constraint concerns relative mining frequencies in blockchain architectures. To apply the pattern, one needs to ensure that it will indeed be highly unlikely that the mining frequency of untrusted nodes exceeds the mining frequency of trusted nodes by the number of confirmation blocks. Otherwise, entries of a blockchain may be subject to modification by untrusted entities and the pattern would fail its guarantee.

7.2.2. Problem 2: Unnecessary Constraints

The support for verification also has the potential to uncover unnecessary constraints in a pattern specification. If certain assumptions a pattern makes about an architecture are not used for the verification of its guarantee, the corresponding constraints can be removed and the scope of the pattern is increased. For example, many descriptions of blockchain architectures require the data entries to be financial transactions with

corresponding private and public keys. However, these assumptions are not required to guarantee persistence of entries and they unnecessarily restrict the application scope of the pattern.

Note, however, that the problem of too strong assumptions, compared to the problem of too weak assumptions, cannot be guaranteed to be solved by verifying the corresponding ADP. A proof of an architectural guarantee may indeed contain unnecessary references to architectural constraints. However, if the proof does not contain any reference to an architectural constraint, the corresponding architectural design constraint can be safely removed from a pattern's specification.

7.3. Limitations

A critical look at the results presented in this paper reveals two major limitations: the lack of support for non-functional aspects and usability of the approach.

7.3.1. Non-functional Aspects

When it comes to ADPs, non-functional aspects play an important role. Many patterns are actually invented to address certain non-functional aspects, such as maintainability. With the approach presented in this paper it is not possible to investigate whether or not a certain pattern really satisfies certain non-functional aspects. Rather, with our approach we focus on the correct implementation of a pattern and we consider them as lemmata to support the verification of architectures using these patterns. Nevertheless, we admit that non-functional aspects play an important role and indeed a lot of research in the architecture community is devoted to this aspect. One line of research uses a quantitative approach and aims towards the development of pattern-specific cost models for certain quality attributes [KKB+99, Mar10]. Another line of research follows a more qualitative approach and uses so-called quality attribute scenarios [BCK07] or grounded theory [GME17] to evaluate quality attributes for patterns.

7.3.2. Target Audience

Using interactive theorem proving make the approach presented in this thesis very general and thus able to address the abstract nature of patterns. However, it makes the approach also difficult to apply, since ITP comes with a steep learning curve and is not yet well-known in the architecture community. The algorithm and its implementation in Eclipse/EMF presented in this paper provide first steps towards making the approach accessible to a broader audience. Moreover, we also provide a calculus to support the interactive verification of patterns in Isabelle/HOL [Mar17b, Mar18a] to support the verification. However, users still need to have some expertise in ITP to efficiently use the approach and thus it might be difficult to apply for practitioners. Thus, as of now, the target group of the approach is mainly researchers in software architectures. In the next section, however, we also provide some ideas for future work to make the approach usable also for practitioners.

7.4. Outlook and Future Work

Figure 55 depicts our overall research agenda in which we envision a *repository* containing a growing collection of verified ADPs. Researchers can connect to the repository and fill it with verification results for existing or even new ADPs. Thereby, they can leverage the hierarchical nature of patterns and verify higher-level patterns using available results from lower level patterns. When verifying an architecture, an architect can connect to the repository and verify the architecture against the assumptions provided by the ADPs. The corresponding guarantee is then automatically transferred to the architecture and can be used to support its verification.

To achieve our vision, future work is required in at least three areas: advanced support for graphical specifications, the development of a pattern verification language, and the integration of our approach into current architecture verification practice.

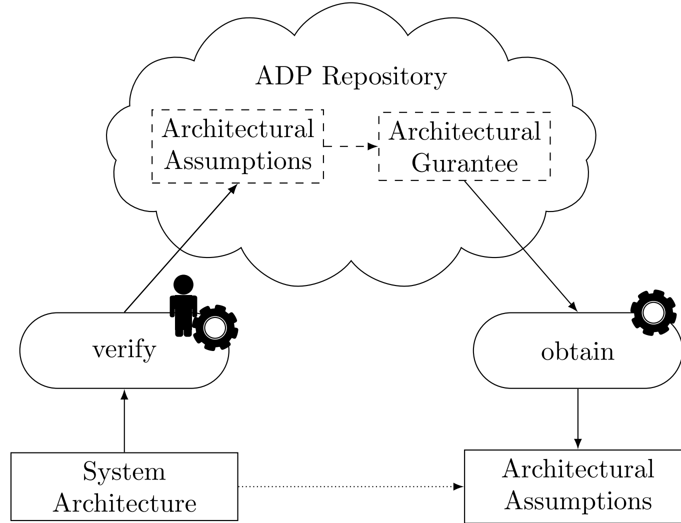


Fig. 55. Research agenda: the repository of verified ADPs.

7.4.1. Advanced Support for Graphical Notation

With this paper, we introduce the notion of architecture diagrams (Sect. 3.4) to support the graphical specification of interfaces and some common architectural constraints. So far, however, expressiveness of graphical annotations is limited to activation and connection constraints. To support the graphical specification of more advanced constraints, future work should investigate possibilities to extend the notion of architecture diagrams. One possible extension, for example, could be the introduction of dependencies between components of a certain type in a notation similar to UML composition. Then, activation and connection annotations could be interpreted relative to such dependencies.

7.4.2. Pattern Verification Language

With this paper, we provide a first step towards interactive pattern verification: An architect can specify a pattern in Eclipse/EMF and then generate a corresponding Isabelle/HOL theory out of it. Then he can verify the pattern in Isabelle/HOL using a corresponding calculus.

However, architects are usually not trained in interactive theorem proving and future work should investigate possibilities to further support an architect in the verification process. A first step could be the development of a more abstract proof language which allows an architect to sketch a proof using abstractions he is familiar with. The abstract proof should then be translated to a corresponding Isabelle/Isar proof and verified by Isabelle.

7.4.3. Integration into Architecture Verification

Another crucial step to achieve our vision concerns the integration of verification results obtained for ADPs to support the verification of architectures. Compared to the verification of ADPs (which can be reused for different architectures), verification of architectures against ADPs has to be done for each architecture, which is why future work should investigate possibilities to automate this step.

7.5. Acknowledgments

We would like to thank *Manfred Broy* and the anonymous reviewers of *FASE 2018* and *Formal Aspects of Computing* for their comments and helpful suggestions on earlier versions of this paper. Moreover, we would like to thank *Dominik Ascher* and *Sebastian Wilzbach* for their valuable support on Eclipse/EMF. The work was partially funded by the *German Federal Ministry of Education and Research (BMBF)* under

grant number “01Is16043A” and the *German Federal Ministry of Economics and Technology (BMWi)* under grant number “0325811A”.

References

[ADG98] Robert Allen, Remi Douence, and David Garlan. Specifying and analyzing dynamic software architectures. In Egidio Astesiano, editor, *Fundamental Approaches to Software Engineering*, volume 1382 of *Lecture Notes in Computer Science*, pages 21–37. Springer Berlin Heidelberg, 1998.

[All97] Robert J Allen. A formal approach to software architecture. Technical report, DTIC Document, 1997.

[AM02a] Nazareno Aguirre and Tom Maibaum. Reasoning about reconfigurable object-based systems in a temporal logic setting. In *Proceedings of IDPT*, 2002.

[AM02b] Nazareno Aguirre and Tom Maibaum. A temporal logic approach to the specification of reconfigurable component-based systems. In *Automated Software Engineering*, pages 271–274. IEEE, 2002.

[Arb04] Farhad Arbab. Reo: a channel-based coordination model for component composition. *Mathematical structures in computer science*, 14(03):329–366, 2004.

[Bal04] Clemens Ballarin. Locales and locale expressions in isabelle/isar. *Lecture notes in computer science*, 3085:34–50, 2004.

[BC13] Yves Bertot and Pierre Castéran. *Interactive theorem proving and program development: Coq'Art: the calculus of inductive constructions*. Springer Science & Business Media, 2013.

[BCK07] Len Bass, Paul Clements, and Rick Kazman. *Software Architecture in Practice*. Addison-Wesley Longman Publishing Co., Inc., 2007.

[Ber96] Klaus Bergner. *Spezifikation großer Objektgeflechte mit Komponentendiagrammen*. PhD thesis, Technische Universität München, 1996.

[Bet16] Lorenzo Bettini. *Implementing domain-specific languages with Xtext and Xtend*. Packt Publishing Ltd, 2016.

[BFGea93] Manfred Broy, Christian Facchi, Radu Grosu, and et al. The requirement and design specification language spectrum – an informal introduction. Technical report, Technische Universität München, 1993.

[BHL+14] Jasmin Christian Blanchette, Johannes Hölzl, Andreas Lochbihler, Lorenz Panny, Andrei Popescu, and Dmitriy Traytel. Truly modular (co) datatypes for isabelle/hol. In *International Conference on Interactive Theorem Proving*, pages 93–110. Springer, 2014.

[BK86] Jan A Bergstra and Jan Willem Klop. Algebra of communicating processes. *CWI Monograph series*, 3:89–138, 1986.

[BMR+96] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. *Pattern-Oriented Software Architecture: A System of Patterns*. Wiley West Sussex, England, 1996.

[Bro96] Manfred Broy. Algebraic specification of reactive systems. In *Algebraic Methodology and Software Technology*, pages 487–503. Springer, Springer Berlin Heidelberg, 1996.

[Bro10] Manfred Broy. A logical basis for component-oriented software and systems engineering. *The Computer Journal*, 53(10):1758–1782, February 2010.

[Bro14] Manfred Broy. A model of dynamic systems. In Saddek Bensalem, Yassine Lakhneck, and Axel Legay, editors, *From Programs to Systems. The Systems Perspective in Computing*, volume 8415 of *Lecture Notes in Computer Science*, pages 39–53. Springer Berlin Heidelberg, 2014.

[BS01] Manfred Broy and Ketil Stolen. *Specification and development of interactive systems: focus on streams, interfaces, and refinement*. Springer Science & Business Media, 2001.

[BSAR06] Christel Baier, Marjan Sirjani, Farhad Arbab, and Jan Rutten. Modeling component connectors in reo by constraint automata. *Science of computer programming*, 61(2):75–113, 2006.

[CAPM10] Pablo F Castro, Nazareno M Aguirre, Carlos Gustavo López Pombo, and Thomas SE Maibaum. Towards managing dynamic reconfiguration of software systems in a categorical setting. In *Lecture Notes in Computer Science*, pages 306–321. Springer, 2010.

[CCGR00] Alessandro Cimatti, Edmund Clarke, Fausto Giunchiglia, and Marco Roveri. Nusmv: a new symbolic model checker. *International Journal on Software Tools for Technology Transfer*, 2(4):410–425, 2000.

[CCS12] Carlos Canal, Javier Cámara, and Gwen Salaün. Structural reconfiguration of systems under behavioral adaptation. *Science of Computer Programming*, 78(1):46 – 64, 2012. Special Section: Formal Aspects of Component Software (FACS'09).

[Cha89] K Mani Chandy. *Parallel program design*. Springer, 1989.

[DVdHT01] Eric M Dashofy, André Van der Hoek, and Richard N Taylor. A highly-extensible, xml-based architecture description language. In *Software Architecture, 2001. Proceedings. Working IEEE/IFIP Conference on*, pages 103–112. IEEE, 2001.

[FLV06] Peter H Feiler, Bruce A Lewis, and Steve Vestal. The sae architecture analysis & design language (aadl) a standard for engineering performance critical systems. In *Computer Aided Control System Design, Control Applications, Intelligent Control*, pages 1206–1211. IEEE, 2006.

[FM97] JoséLuiz Fiadeiro and Tom Maibaum. Categorical semantics of parallel program design. *Science of Computer Programming*, 28(2-3):111–138, 1997.

[FS97] D. Fensel and A. Schnogge. Using kiv to specify and verify architectures of knowledge-based systems. In *Automated Software Engineering*, pages 71–80, November 1997.

[Gar03] David Garlan. Formal modeling and analysis of software architecture: Components, connectors, and events. In *Formal Methods for Software Architectures*, pages 1–24. Springer, 2003.

[GH05] Jeremy Gibbons and Graham Hutton. Proof methods for corecursive programs. *Fundam. Inform.*, 66:353–366, 2005.

[GHJV94] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. Design patterns: Elements of reusable object-oriented software, 1994.

[GJS17] Thomas Göthel, Nils Jähnig, and Simon Seif. Refinement-based modelling and verification of design patterns for self-adaptive systems. In *International Conference on Formal Engineering Methods*, pages 157–173. Springer, 2017.

[GM18] Habtom Kahsay Gidey and Diego Marmsoler. FACTum Studio. `https://habtom.github.io/factum/`, 2018.

[GME17] H. K. Gidey, D. Marmsoler, and J. Eckhardt. Grounded architectures: Using grounded theory for the design of software architectures. In *2017 IEEE International Conference on Software Architecture Workshops (ICSAW)*, pages 141–148, April 2017.

[GMW00] David Garlan, Robert T Monroe, and David Wile. Acme: Architectural description of component-based systems. *Foundations of component-based systems*, 68:47–68, 2000.

[GR91] Michael M. Gorlick and Rami R. Razouk. Using weaves for software construction and analysis. In Les Belady, David R. Barstow, and Koji Torii, editors, *Proceedings of the 13th International Conference on Software Engineering, Austin, TX, USA, May 13-17, 1991.*, pages 23–34. IEEE Computer Society / ACM Press, 1991.

[GRABR14] Thomas Gibson-Robinson, Philip Armstrong, Alexandre Boulgakov, and Andrew W Roscoe. Fdr3—a modern refinement checker for csp. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 187–201. Springer, 2014.

[HF10] Florian Hölzl and Martin Feilkas. Autofocus 3: A scientific tool prototype for model-based development of component-based, reactive, distributed systems. In *Proceedings of the 2007 International Dagstuhl Conference on Model-based Engineering of Embedded Real-time Systems*, MBEERTS'07, pages 317–322, Berlin, Heidelberg, 2010. Springer-Verlag.

[Hoa78] Charles Antony Richard Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, 1978.

[Jac02] Daniel Jackson. Alloy: A lightweight object modelling notation. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 11(2):256–290, 2002.

[JR97] Bart Jacobs and Jan Rutten. A tutorial on (co)algebras and (co)induction. *EATCS Bulletin*, 62:62–222, 1997.

[KG06] Jung Soo Kim and David Garlan. Analyzing architectural styles with alloy. In *Proceedings of the ISSTA 2006 workshop on Role of software architecture for testing and analysis*, pages 70–80. ACM, 2006.

[KKB+99] Mark H Klein, Rick Kazman, Len Bass, Jeromy Carriere, Mario Barbacci, and Howard Lipson. Attribute-based architecture styles. In *Software Architecture*, pages 225–243. Springer, 1999.

[KMLA11] Christian Krause, Ziyan Maraikar, Alexander Lazovik, and Farhad Arbab. Modeling dynamic reconfigurations in reo using high-level replacement systems. *Science of Computer Programming*, 76(1):23 – 36, 2011. Selected papers from the 6th International Workshop on the Foundations of Coordination Languages and Software Architectures.

[KRDO17] Aggelos Kiayias, Alexander Russell, Bernardo David, and Roman Oliynykov. Ouroboros: A provably secure proof-of-stake blockchain protocol. In *Annual International Cryptology Conference*, pages 357–388. Springer, 2017.

[LKA+95] David C Luckham, John J Kenney, Larry M Augustin, James Vera, Doug Bryan, and Walter Mann. Specification and analysis of system architecture using rapide. *Software Engineering, IEEE Transactions on*, 21(4):336–354, 1995.

[LMP10] Francois Laroussinie, Antoine Meyer, and Eudes Petonnet. Counting LTL. In *2010 17th International Symposium on Temporal Representation and Reasoning*. IEEE, sep 2010.

[Loc10] Andreas Lochbihler. Coinduction. *The Archive of Formal Proof s. http://afp.sourceforge.net/entries/Coinductive.shtml*, 2010.

[LS13] Yi Li and Meng Sun. Modeling and analysis of component connectors in coq. In José Luiz Fiadeiro, Zhiming Liu, and Jinyun Xue, editors, *Formal Aspects of Component Software - 10th International Symposium, FACS 2013, Nanchang, China, October 27-29, 2013, Revised Selected Papers*, volume 8348 of *Lecture Notes in Computer Science*, pages 273–290. Springer, 2013.

[Mar10] Diego Marmsoler. Applying the scientific method in the definition and analysis of a new architectural style. Master's thesis, Free University of Bolzano-Bozen, 2010.

[Mar17a] Diego Marmsoler. Dynamic architectures. *Archive of Formal Proofs*, July 2017. `http://isa-afp.org/entries/DynamicArchitectures.html`, Formal proof development.

[Mar17b] Diego Marmsoler. Towards a calculus for dynamic architectures. In Dang Van Hung and Deepak 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.

[Mar18a] Diego Marmsoler. A framework for interactive verification of architectural design patterns in isabelle/hol. In *The 20th International Conference on Formal Engineering Methods, ICFEM 2018, Proceedings*, 2018.

[Mar18b] Diego Marmsoler. A theory of architectural design patterns. *Archive of Formal Proofs*, March 2018. `http://isa-afp.org/entries/Architectural_Design_Patterns.html`, Formal proof development.

[MCL04] Jeffrey KH Mak, Clifford ST Choy, and Daniel PK Lun. Precise modeling of design patterns in uml. In *Software Engineering*, pages 252–261. IEEE, 2004.

[MD17] Diego Marmsoler and Silvio Degenhardt. Verifying patterns of dynamic architectures using model checking. In *Proceedings International Workshop on Formal Engineering approaches to Software Components and Architectures, FESCA@ETAPS 2017, Uppsala, Sweden, 22nd April 2017.*, pages 16–30, 2017.

[MG16a] D. Marmsoler and M. Gleirscher. On activation, connection, and behavior in dynamic architectures. *Scientific Annals of Computer Science*, 26(2):187–248, 2016.

[MG16b] Diego Marmsoler and Mario Gleirscher. Specifying properties of dynamic architectures using configuration traces. In *International Colloquium on Theoretical Aspects of Computing*, pages 235–254. Springer, 2016.

[MG18] Diego Marmsoler and Habtom Kahsay Gidey. Factum studio: A tool for the axiomatic specification and verification of architectural design patterns. In *Formal Aspects of Component Software - FACS 2018 - 15th International Conference, Proceedings*, 2018.

[Mil99] Robin Milner. *Communicating and Mobile Systems: the π-calculus*. Cambridge university press, 1999.

[MK96] Jeff Magee and Jeff Kramer. Dynamic structure in software architectures. In David 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.

[MP92] Zohar Manna and Amir Pnueli. *The Temporal Logic of Reactive and Concurrent Systems*. Springer New York, 1992.

[Nak08] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system, 2008.

[NPW02] Tobias Nipkow, Lawrence C Paulson, and Markus Wenzel. *Isabelle/HOL: a proof assistant for higher-order logic*, volume 2283. Springer Science & Business Media, 2002.

[Oqu04] Flavio Oquendo. π-adl: An architecture description language based on the higher-order typed π-calculus for specifying dynamic and mobile software architectures. *ACM SIGSOFT Software Engineering Notes*, 29(3):1–14, May 2004.

[Rau01] Andreas Rausch. *Componentware*. Dissertation, Technische Universität München, München, 2001.

[Rei95] Wolfgang Reif. The kiv-approach to software verification. *KORSO: Methods, Languages, and Tools for the Construction of Correct Software*, pages 339–368, 1995.

[RJB04] James Rumbaugh, Ivar Jacobson, and Grady Booch. *Unified modeling language reference manual, the*. Pearson Higher Education, 2004.

[SBR12] Alejandro Sanchez, Luís Soares. Barbosa, and Daniel Riesco. Bigraphical modelling of architectural patterns. In Farhad Arbab and Peter Csaba Ölveczky, editors, *Formal Aspects of Component Software*, pages 313–330, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[SG96] Mary Shaw and David Garlan. *Software Architecture: Perspectives on an Emerging Discipline*, volume 1. Prentice Hall Englewood Cliffs, 1996.

[SH04] Neelam Soundarajan and Jason O Hallstrom. Responsibilities and rewards: Specifying design patterns. In *Software Engineering*, pages 666–675. IEEE, 2004.

[SMB15] Alejandro Sanchez, Alexandre Madeira, and Luís S Barbosa. On the verification of architectural reconfigurations. *Computer Languages, Systems & Structures*, 44:218–237, 2015.

[Spi07] Maria Spichkova. *Specification and seamless verification of embedded real-time systems: FOCUS on Isabelle*. PhD thesis, Technical University Munich, Germany, 2007.

[TMD09] Richard N Taylor, Nenad Medvidovic, and Eric M Dashofy. *Software Architecture: Foundations, Theory, and Practice*. Wiley Publishing, 2009.

[TO17] TypeFox and Obeo. Xtext / sirius - integration the main use-cases. `https://goo.gl/8bcWJc`, 2017.

[vOvdLKM00] Rob C. van Ommering, Frank van der Linden, Jeff Kramer, and Jeff Magee. The koala component model for consumer electronics software. *IEEE Computer*, 33(3):78–85, 2000.

[W+04] Makarius Wenzel et al. The isabelle/isar reference manual, 2004.

[Wen07] Makarius Wenzel. Isabelle/isar – a generic framework for human-readable proof documents. *From Insight to Proof – Festschrift in Honour of Andrzej Trybulec*, 10(23):277–298, 2007.

[WF02] Michel Wermelinger and José Luiz Fiadeiro. A graph transformation approach to software architecture reconfiguration. *Science of Computer Programming*, 44(2):133 – 155, 2002. Special Issue on Applications of Graph Transformations (GRATRA 2000).

[Wir90] Martin Wirsing. Algebraic specification. In Jan van Leeuwen, editor, *Handbook of Theoretical Computer Science (Vol. B)*, pages 675–788. MIT Press, Cambridge, MA, USA, 1990.

[WLF01] Michel Wermelinger, Antónia Lopes, and José Luiz Fiadeiro. A graph based architectural (re)configuration language. In *Software Engineering Notes*, volume 26, pages 21–32. ACM, 2001.

[WSWS08] Stephen Wong, Jing Sun, Ian Warren, and Jun Sun. A scalable approach to multi-style architectural modeling and verification. In *Engineering of Complex Computer Systems*, pages 25–34. IEEE, 2008.

[ZA05] Uwe Zdun and Paris Avgeriou. Modeling architectural patterns using architectural primitives. In Ralph E. Johnson and Richard P. Gabriel, editors, *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2005, October 16-20, 2005, San Diego, CA, USA*, pages 133–146. ACM, 2005.

[ZLS+12] Jiexin Zhang, Yang Liu, Jing Sun, Jin Song Dong, and Jun Sun. Model checking software architecture design. In *High-Assurance Systems Engineering*, pages 193–200. IEEE, 2012.

Behavior terms: syntax

$$v \in DV_s \implies \text{“}v\text{”} \in {}^{s}_{\Sigma}BT_{DV}(ps) \ ,$$
$$p \in PID \implies \text{“}p\text{”} \in {}^{s}_{\Sigma}BT_{DV}(ps)[\text{for } \mathsf{tp}(p) = s] \ ,$$
$$f \in F^0 \implies \text{“}f\text{”} \in {}^{s}_{\Sigma}BT_{DV}(ps) \ [\text{for } \mathsf{sf}(f)_{(0)} = s] \ ,$$
$$\left.\begin{array}{l} f \in F^{n+1} \quad \wedge \\ \text{“}t_1\text{”} \in {}^{s_1}_{\Sigma}BT_{DV}(ps), \cdots, \\ \text{“}t_{n+1}\text{”} \in {}^{s_{n+1}}_{\Sigma}BT_{DV}(ps) \end{array}\right\} \implies \left\{\begin{array}{l} \text{“}f(t_1, \cdots, t_{n+1})\text{”} \in {}^{s}_{\Sigma}BT_{DV}(ps) \\ [\text{for } n \in \mathbb{N}, \mathsf{sf}(f)_{(0)} = s, \text{ and} \\ \mathsf{sf}(f)_{(1)} = s_1, \cdots, \mathsf{sf}(f)_{(n+1)} = s_{n+1}] \ . \end{array}\right.$$

Fig. 56. Inductive definition of behavior terms.

A. Behavior Trace Assertions

Behavior trace assertions are formulated over data type variables, i.e., variables representing messages of a certain type. Thus, given a signature $\Sigma = (S, F, B)$, we assume the existence of a family of data type variables $DV = (DV_s)_{s \in S}$ and rigid data type variables DV'. Both types of variables are interpreted over an algebra $A = \big((A_s)_{s\in S}, (f^A)_{f\in F}, (p^A)_{p\in B}\big) \in \mathcal{A}(\Sigma)$ for signature Σ (where F^n and B^n denote all the function/predicate symbols of arity n, and sf and sp assign a tuple of sorts to each function/predicate symbol, respectively[6]). Thereby, data type variable assignments $\iota = (\iota_s)_{s\in S}$ consist of interpretations $\iota_s \colon DV_s \to A_s$, which are newly evaluated at each point in time. Rigid data type variables, on the other hand, are interpreted only once for the whole execution by a so-called rigid data type variable assignment $\iota' = (\iota'_s)_{s\in S}$. With $\mathcal{I}^{DV}_A$ we denote the set of all data type variable assignments for data type variables DV in algebra A and with $\mathcal{I}'^{DV'}_A$ the set of all rigid data type variable assignments for rigid data type variables DV' in algebra A, respectively.

A.1. Behavior terms

A.1.1. Syntax

Definition 9 (Behavior terms: syntax). The set of all *behavior terms* of sort $s \in S$ over a signature $\Sigma = (S, F, B)$, datatype variables DV, and port specification $ps = (PID, \mathsf{tp})$, is the smallest set ${}^{s}_{\Sigma}BT_{DV}(ps)$ satisfying the equations of Fig. 56. The set of all behavior terms of all sorts is denoted by ${}_{\Sigma}BT_{DV}(ps)$.

A.1.2. Semantics

Definition 10 (Behavior terms: semantics). The semantics of behavior terms ${}_{\Sigma}BT_{DV}(ps)$, formulated over port specification $ps = (PID, \mathsf{tp})$, is defined over an algebra $A \in \mathcal{A}(\Sigma)$ with corresponding data type variable assignments $\iota \in \mathcal{I}^{DV}_A$ and a valuation $\mu \in \overline{\mathcal{P}}$ of a set of ports $\mathcal{P}$ with corresponding interpretation $\delta \colon PID \to \mathcal{P}$ for the port identifiers of ps. It is given by a semantic function ${}^{\iota}_{A}[\![_]\!]^{\delta}_{\mu} \colon \overrightarrow{\bigcup}_{s\in S}({}^{s}_{\Sigma}BT_{DV}(ps) \to A_s)$, defined recursively by the equations provided in Fig. 57.

A.2. Behavior assertions

A.2.1. Syntax

Definition 11 (Behavior assertions: syntax). The set of all *behavior assertions* over a signature $\Sigma = (S, F, B)$, datatype variables DV, and port specification $ps = (PID, \mathsf{tp})$, is the smallest set ${}_{\Sigma}BA_{DV}(ps)$ satisfying the equations of Fig. 58.

---

[6] For function symbols, the sort for the return type is assumed to be on position 0 of the tuple.

Behavior terms: semantics

$$
\begin{aligned}
{}^{\iota}_{A}[\![\text{“}v\text{”}]\!]^{\delta}_{\mu} &= \iota_s(v) \text{ [for } v \in DV_s] \ , \\
{}^{\iota}_{A}[\![\text{“}p\text{”}]\!]^{\delta}_{\mu} &= \mu(\delta(p)) \text{ [for } p \in PID] \ , \\
{}^{\iota}_{A}[\![\text{“}f\text{”}]\!]^{\delta}_{\mu} &= A_f \text{ [for function symbol } f \in F^0] \ , \\
{}^{\iota}_{A}[\![\text{“}f(t_1, \cdots, t_n)\text{”}]\!]^{\delta}_{\mu} &= \begin{cases} A_f\left({}^{\iota}_{A}[\![\text{“}t_1\text{”}]\!]^{\delta}_{\mu}, \cdots, {}^{\iota}_{A}[\![\text{“}t_n\text{”}]\!]^{\delta}_{\mu}\right) \\ \text{[for function symbol } f \in F^{n+1}] \ . \end{cases}
\end{aligned}
$$

Fig. 57. Recursive definition of semantic function for behavior terms.

Behavior assertions: syntax

$$
\begin{aligned}
& & & \text{“}\mathit{true}\text{”} \in {}_{\Sigma}BA_{DV}(ps) \ , \\
& & & \text{“}\mathit{false}\text{”} \in {}_{\Sigma}BA_{DV}(ps) \ , \\
& b \in B^0 & \Longrightarrow \ & \text{“}b\text{”} \in {}_{\Sigma}BA_{DV}(ps) \ , \\
& \left.\begin{array}{l} b \in B^{n+1} \quad \wedge \\ \text{“}t_1\text{”} \in {}^{s_1}_{\Sigma}BT_{DV}(ps), \cdots, \\ \text{“}t_{n+1}\text{”} \in {}^{s_{n+1}}_{\Sigma}BT_{DV}(ps) \end{array}\right\} & \Longrightarrow \ & \begin{cases} \text{“}b(t_1, \cdots, t_{n+1})\text{”} \in {}_{\Sigma}BA_{DV}(ps) \\ \text{[for } n \in \mathbb{N} \text{ and } \mathsf{sp}(b)_{(1)} = s_1, \cdots, \mathsf{sp}(b)_{(n+1)} = s_{n+1}] \ , \end{cases} \\
& \text{“}t\text{”}, \text{“}t'\text{”} \in {}^{s}_{\Sigma}BT_{DV}(ps) & \Longrightarrow \ & \text{“}t = t'\text{”} \in {}_{\Sigma}BA_{DV}(ps) \text{ [for some } s \in S] \ , \\
& \text{“}\phi\text{”} \in {}_{\Sigma}BA_{DV}(ps) & \Longrightarrow \ & \text{“}\neg\phi\text{”} \in {}_{\Sigma}BA_{DV}(ps) \ , \\
& \text{“}\phi\text{”}, \text{“}\phi'\text{”} \in {}_{\Sigma}BA_{DV}(ps) & \Longrightarrow \ & \begin{cases} \text{“}\phi \wedge \phi'\text{”} \in {}_{\Sigma}BA_{DV}(ps), \\ \text{“}\phi \vee \phi'\text{”} \in {}_{\Sigma}BA_{DV}(ps), \\ \text{“}\phi \longrightarrow \phi'\text{”} \in {}_{\Sigma}BA_{DV}(ps), \\ \text{“}\phi \longleftrightarrow \phi'\text{”} \in {}_{\Sigma}BA_{DV}(ps). \ , \end{cases} \\
& \text{“}\phi\text{”} \in {}_{\Sigma}BA_{DV}(ps) \wedge x \in DV_s & \Longrightarrow \ & \begin{cases} \text{“}\forall x\colon\ \phi\text{”} \in {}_{\Sigma}BA_{DV}(ps), \\ \text{“}\exists x\colon\ \phi\text{”} \in {}_{\Sigma}BA_{DV}(ps) \text{ [for } s \in S]. \ . \end{cases}
\end{aligned}
$$

Fig. 58. Inductive definition of behavior assertions.

A.2.2. Semantics

Definition 12 (Behavior assertions: semantics). The semantics of *behavior assertions* ${}_{\Sigma}BA_{DV}(ps)$, formulated over port specification $ps = (PID, \mathsf{tp})$, is defined over an algebra $A \in \mathcal{A}(\Sigma)$ with corresponding data type variable assignments $\iota \in \mathcal{I}_A^{DV}$ and an interpretation $\delta\colon PID \to \mathcal{P}$ for the port identifiers of ps with concrete ports of a set $\mathcal{P}$. It is given by a relation $\models^{\delta}_{A,\iota} \subseteq \overline{\mathcal{P}} \times {}_{\Sigma}BA_{DV}(ps)$ defined recursively by the equations provided in Fig. 59

A.3. Behavior trace assertions

A.3.1. Syntax

Definition 13 (Behavior trace assertions: syntax). The set of all *behavior trace assertions* over a signature $\Sigma = (S, F, B)$, *disjoint* sets of datatype variables DV and rigid datatype variables DV', and port specification $ps = (PID, \mathsf{tp})$, is the smallest set ${}_{\Sigma}BTA_{DV}^{DV'}(ps)$ satisfying the equations of Fig. 60.

Behavior assertions: semantics

$$
\begin{array}{rcl}
\mu \models^{\delta}_{A,\iota} \text{“}\mathit{true}\text{”} & & , \\
\neg\big(\mu \models^{\delta}_{A,\iota} \text{“}\mathit{false}\text{”}\big) & & , \\
\mu \models^{\delta}_{A,\iota} \text{“}b\text{”} & \iff & A_b \ [\text{for } b \in B^0] \ , \\
\mu \models^{\delta}_{A,\iota} \text{“}b(t_1, \cdots, t_n)\text{”} & \iff & A_b\big({}^{\iota}_{A}[\![\text{“}t_1\text{”}]\!]^{\delta}_{\mu}, \cdots, {}^{\iota}_{A}[\![\text{“}t_n\text{”}]\!]^{\delta}_{\mu}\big) \ [\text{for } b \in B^{n+1}] \ , \\
\mu \models^{\delta}_{A,\iota} \text{“}t = t'\text{”} & \iff & {}^{\iota}_{A}[\![\text{“}t\text{”}]\!]^{\delta}_{\mu} = {}^{\iota}_{A}[\![\text{“}t'\text{”}]\!]^{\delta}_{\mu} \ , \\
\mu \models^{\delta}_{A,\iota} \text{“}\phi \wedge \phi & \iff & \mu \models^{\delta}_{A,\iota} \text{“}\phi\text{”} \wedge \mu \models^{\delta}_{A,\iota} \text{“}\phi'\text{”} \ , \\
\mu \models^{\delta}_{A,\iota} \text{“}\phi \vee \phi'\text{”} & \iff & \mu \models^{\delta}_{A,\iota} \text{“}\phi\text{”} \wedge \mu \models^{\delta}_{A,\iota} \text{“}\phi'\text{”} \ , \\
\mu \models^{\delta}_{A,\iota} \text{“}\phi \longrightarrow \phi'\text{”} & \iff & \mu \models^{\delta}_{A,\iota} \text{“}\phi\text{”} \wedge \mu \models^{\delta}_{A,\iota} \text{“}\phi'\text{”} \ , \\
\mu \models^{\delta}_{A,\iota} \text{“}\phi \longleftrightarrow \phi'\text{”} & \iff & \mu \models^{\delta}_{A,\iota} \text{“}\phi\text{”} \wedge \mu \models^{\delta}_{A,\iota} \text{“}\phi'\text{”} \ , \\
\mu \models^{\delta}_{A,\iota} \text{“}\exists x\colon\ \phi\text{”} & \iff & \begin{cases} \exists x' \in A_s \colon \mu \models^{\delta}_{A,\iota[s\colon x \mapsto x']} \text{“}\phi\text{”} \\ [\text{for } s \in S \text{ and } x \in DV_s] \ , \end{cases} \\
\mu \models^{\delta}_{A,\iota} \text{“}\forall x\colon\ \phi\text{”} & \iff & \begin{cases} \forall x' \in A_s \colon \mu \models^{\delta}_{A,\iota[s\colon x \mapsto x']} \text{“}\phi\text{”} \\ [\text{for } s \in S \text{ and } x \in DV_s] \ , \end{cases}
\end{array}
$$

Fig. 59. Recursive definition of satisfaction relation for behavior assertions.

Behavior trace assertions: syntax

$$
\begin{array}{rcl}
 & & \text{“}\mathit{true}\text{”},\ \text{“}\mathit{false}\text{”} \in {}_{\Sigma}BTA^{DV'}_{DV}(ps), \\
\phi \in {}_{\Sigma}BA_{DV \cup DV'}(ps) & \Longrightarrow & \phi \in {}_{\Sigma}BTA^{DV'}_{DV}(ps), \\
\text{“}\gamma\text{”} \in {}_{\Sigma}BTA^{DV'}_{DV}(ps) & \Longrightarrow & \text{“}\neg\gamma\text{”},\ \text{“}\bigcirc\gamma\text{”},\ \text{“}\Diamond\gamma\text{”},\ \text{“}\Box\gamma\text{”} \in {}_{\Sigma}BTA^{DV'}_{DV}(ps), \\
\text{“}\gamma\text{”}, \text{“}\gamma'\text{”} \in {}_{\Sigma}BTA^{DV'}_{DV}(ps) & \Longrightarrow & \begin{cases} \text{“}\gamma \wedge \gamma'\text{”},\ \text{“}\gamma \vee \gamma'\text{”}, \\ \text{“}\gamma \longrightarrow \gamma'\text{”},\ \text{“}\big(\gamma'\ \mathcal{U}\ \gamma\big)\text{”} \in {}_{\Sigma}BTA^{DV'}_{DV}(ps), \end{cases} \\
\left.\begin{array}{r} x \in DV' \ \wedge \\ \text{“}\gamma\text{”} \in {}_{\Sigma}BTA^{DV'}_{DV}(ps) \end{array}\right\} & \Longrightarrow & \begin{cases} \text{“}\forall x\colon\ \gamma\text{”} \in {}_{\Sigma}BTA^{DV'}_{DV}(ps), \\ \text{“}\exists x\colon\ \gamma\text{”} \in {}_{\Sigma}BTA^{DV'}_{DV}(ps). \end{cases}
\end{array}
$$

Fig. 60. Inductive definition of behavior trace assertions.

A.3.2. Semantics

Definition 14 (Behavior trace assertions: semantics). The semantics of behavior trace assertions ${}_{\Sigma}BTA^{DV'}_{DV}(ps)$, formulated over port specification $ps = (PID, \mathtt{tp})$, is defined over an algebra $A \in \mathcal{A}(\Sigma)$ with corresponding rigid data type variable assignments $\iota' \in \mathcal{I'}^{DV'}_{A}$ and an interpretation $\delta\colon PID \to \mathcal{P}$ for the port identifiers of ps with concrete ports of a set $\mathcal{P}$. It is given by a relation $\models^{\delta}_{A,\iota'} \subseteq \big((\overline{\mathcal{P}})^{\infty} \times \mathbb{N}\big) \times {}_{\Sigma}BA_{DV}(ps)$ defined recursively by the equations provided in Fig. 61.

Behavior trace assertions: semantics

$$
\begin{array}{rcl}
(t,n) \models^{\delta}_{A,\iota'} \text{“}\mathit{true}\text{”} & & , \\
\neg\big((t,n) \models^{\delta}_{A,\iota'} \text{“}\mathit{false}\text{”}\big) & & , \\
(t,n) \models^{\delta}_{A,\iota'} \phi & \iff & \forall \iota \in \mathcal{I}^{DV}_{A} : t(n) \models_{A,\iota\cup\iota'} \phi \text{ [for } \phi \in {}_{\Sigma}BA_{DV}(ps)], \\
(t,n) \models^{\delta}_{A,\iota'} \text{“}\bigcirc\gamma\text{”} & \iff & (t,n+1) \models^{\delta}_{A,\iota'} \text{“}\gamma\text{”}, \\
(t,n) \models^{\delta}_{A,\iota'} \text{“}\Diamond\gamma\text{”} & \iff & \exists n' \geq n \colon (t,n') \models^{\delta}_{A,\iota'} \text{“}\gamma\text{”}, \\
(t,n) \models^{\delta}_{A,\iota'} \text{“}\Box\gamma\text{”} & \iff & \forall n' \geq n \colon (t,n') \models^{\delta}_{A,\iota'} \text{“}\gamma\text{”}, \\
(t,n) \models^{\delta}_{A,\iota'} \text{“}\gamma' \,\mathcal{U}\, \gamma\text{”} & \iff & \begin{cases} \exists n' \geq n \colon (t,n') \models^{\delta}_{A,\iota'} \text{“}\gamma\text{”} \;\wedge \\ \forall n \leq m < n' \colon (t,m) \models^{\delta}_{A,\iota'} \text{“}\gamma'\text{”}, \end{cases} \\
(t,n) \models_{A,\iota'} \text{“}\exists x \colon\; \gamma\text{”} & \iff & \begin{cases} \exists x' \in A_s \colon (t,n) \models^{\delta}_{A,\iota'[s\colon x \mapsto x']} \text{“}\gamma\text{”} \\ \text{[for } s \in S \text{ and } x \in DV_s] \;, \end{cases} \\
(t,n) \models_{A,\iota'} \text{“}\forall x \colon\; \gamma\text{”} & \iff & \begin{cases} \forall x' \in A_s \colon (t,n) \models^{\delta}_{A,\iota'[s\colon x \mapsto x']} \text{“}\gamma\text{”} \\ \text{[for } s \in S \text{ and } x \in DV_s] \;. \end{cases}
\end{array}
$$

Fig. 61. Recursive definition of satisfaction relation for behavior trace assertions.

Architecture terms: syntax

$$v \in DV_s \implies \text{“}v\text{”} \in {}^{s}_{\Sigma}CT^{DV}_{CV}(is) \ ,$$

$$v \in (CV_i)_\omega \wedge p \in \mathsf{port}(\mathsf{if}(i)) \implies \begin{cases} \text{“}v.p\text{”} \in {}^{s}_{\Sigma}CT^{DV}_{CV}(is) \\ [\text{for } i \in I \text{ and } \mathsf{tp}(p) = s] \ , \end{cases}$$

$$f \in F^0 \implies \text{“}f\text{”} \in {}^{s}_{\Sigma}CT^{DV}_{CV}(is) \ [\text{for } \mathsf{sf}(f)_{(0)} = s] \ ,$$

$$\left.\begin{array}{l} f \in F^{n+1} \quad \wedge \\ \text{“}t_1\text{”} \in {}^{s_1}_{\Sigma}CT^{DV}_{CV}(is), \cdots, \\ \text{“}t_{n+1}\text{”} \in {}^{s_{n+1}}_{\Sigma}CT^{DV}_{CV}(is) \end{array}\right\} \implies \begin{cases} \text{“}f(t_1, \cdots, t_{n+1})\text{”} \in {}^{s}_{\Sigma}CT^{DV}_{CV}(is) \\ [\text{for } n \in \mathbb{N}, \ \mathsf{sf}(f)_{(0)} = s, \text{ and} \\ \mathsf{sf}(f)_{(1)} = s_1, \cdots, \mathsf{sf}(f)_{(n+1)} = s_{n+1}] \ . \end{cases}$$

Fig. 62. Inductive definition of architecture terms.

B. Architecture Trace Assertions

In addition to variables for data types (as introduced already for behavior trace assertions), architecture trace assertions are formulated also over component variables, i.e., variables representing components of a certain type. Thus, given a signature Σ and an interface specification $is = (I, \mathsf{if})$ over port specification (PID, tp), we assume the existence of a family of component variables $CV = (CV_i)_{i \in I}$ with component variables $CV_i = \big((CV_i)_\omega\big)_{\omega\colon p \mapsto DtT_{\mathsf{tp}(p)}(\Sigma, DV)}$ for each interface $i \in I$ and each valuation of component parameters ω. In addition, we assume the existence of a corresponding family of rigid component variables CV'.

Component variables are interpreted over a family of components $\mathcal{C} = (\mathcal{C}_{ct})_{ct \in CT_{\mathcal{I}}}$ by a so-called component variable assignment $\kappa = (\kappa_i)_{i \in I}$, with $\kappa_i = \big((\kappa_i)_\omega\big)_{\omega\colon p \mapsto DtT_{\mathsf{tp}(p)}(\Sigma, DV)}$ and $\kappa_i = (\kappa_i)_\omega \colon (CV_i)_\omega \to \mathcal{C}_{\epsilon(i), \lambda p \in \mathsf{par}(\mathsf{if}(i))\colon {}_A[\![\omega(p)]\!]}$ (for a given interface interpretation ϵ and port interpretation δ). Again, we denote with κ' a corresponding rigid component variable assignment for rigid component variables. The set of all component variable assignments is denoted with $\mathcal{K}^{CV}_{\mathcal{C}}$ and the set of all rigid component variable assignments with $\mathcal{K}'^{CV'}_{\mathcal{C}}$, respectively.

B.1. Architecture Terms

B.1.1. Syntax

Definition 15 (Architecture terms: syntax). The set of all *architecture terms* of sort $s \in S$ over a signature $\Sigma = (S, F, B)$, interface specification $is = (I, \mathsf{if})$ over port specification (PID, tp), datatype variables DV, and component variables CV, is the smallest set ${}^{s}_{\Sigma}CT^{DV}_{CV}(is)$, satisfying the equations of Fig. 62. The set of all architecture terms of all sorts is denoted by ${}_{\Sigma}CT^{DV}_{CV}(is)$.

B.1.2. Semantics

Definition 16 (Architecture terms: semantics). The *semantics* of architecture terms ${}_{\Sigma}CT^{DV}_{CV}(is)$, formulated over interface specification $is = (I, \mathsf{if})$ and port specification (PID, tp), is defined over an algebra $A \in \mathcal{A}(\Sigma)$ with corresponding data type variable assignments $\iota \in \mathcal{I}^{DV}_A$, an architecture snapshot $as \in AS^{\mathcal{C}}_{\mathcal{T}}$ with corresponding port interpretation $\delta\colon PID \to \mathcal{P}$ for the port identifiers of ps, and component interpretation $\kappa \in \mathcal{K}^{CV}_{\mathcal{C}}$. It is given by a semantic function ${}^{\iota}_{A}[\![_]\!]^{\kappa}_{\delta}(as)\colon \overrightarrow{\bigcup}_{s \in S}({}^{s}_{\Sigma}CT^{DV}_{CV}(is) \to A_s)$, defined recursively by the equations provided in Fig. 63.

Architecture terms: semantics

$$
\begin{aligned}
{}^{\iota}_{A}[\![\text{“}v\text{”}]\!]^{\kappa}_{\delta}(as) &= \iota_s(v) \text{ [for } v \in DV_s] \ , \\
{}^{\iota}_{A}[\![\text{“}v.p\text{”}]\!]^{\kappa}_{\delta}(as) &= \begin{cases} val_{as}\Big(\big((\kappa_i)_\omega(v), \big(\delta(p)\big)\big)\Big) \\ \text{[for } i \in I \text{ and } v \in (CV_i)_\omega] \ , \end{cases} \\
{}^{\iota}_{A}[\![\text{“}f\text{”}]\!]^{\kappa}_{\delta}(as) &= A_f \text{ [for function symbol } f \in F^0] \ , \\
{}^{\iota}_{A}[\![\text{“}f(t_1, \cdots, t_n)\text{”}]\!]^{\kappa}_{\delta}(as) &= \begin{cases} A_f\big({}^{\iota}_{A}[\![\text{“}t_1\text{”}]\!]^{\kappa}_{\delta}(as), \cdots, {}^{\iota}_{A}[\![\text{“}t_n\text{”}]\!]^{\kappa}_{\delta}(as)\big) \\ \text{[for function symbol } f \in F^{n+1}] \ . \end{cases}
\end{aligned}
$$

Fig. 63. Recursive definition of semantic function for architecture terms.

Architecture assertions: syntax

$$
\begin{aligned}
& & & \text{“}true\text{”} \in {}_{\Sigma}CA^{DV}_{CV}(is) \ , \\
& & & \text{“}false\text{”} \in {}_{\Sigma}CA^{DV}_{CV}(is) \ , \\
b \in B^0 & &\Longrightarrow\ & \text{“}b\text{”} \in {}_{\Sigma}CA^{DV}_{CV}(is) \ , \\
\left.\begin{aligned} & b \in B^{n+1} \quad \wedge \\ & \text{“}t_1\text{”} \in {}^{s_1}_{\Sigma}CT^{DV}_{CV}(is), \cdots, \\ & \text{“}t_{n+1}\text{”} \in {}^{s_{n+1}}_{\Sigma}CT^{DV}_{CV}(is) \end{aligned}\right\} & &\Longrightarrow\ & \begin{cases} \text{“}b(t_1, \cdots, t_{n+1})\text{”} \in {}_{\Sigma}CA^{DV}_{CV}(is) \\ \text{[for } n \in \mathbb{N} \text{ and} \\ \mathsf{sp}(b)_{(1)} = s_1, \cdots, \mathsf{sp}(b)_{(n+1)} = s_{n+1}] \ , \end{cases} \\
\text{“}t\text{”}, \text{“}t'\text{”} \in {}^{s}_{\Sigma}CT^{DV}_{CV}(is) & &\Longrightarrow\ & \text{“}t = t'\text{”} \in {}_{\Sigma}CA^{DV}_{CV}(is) \text{ [for some } s \in S] \ , \\
\text{“}\phi\text{”} \in {}_{\Sigma}CA^{DV}_{CV}(is) & &\Longrightarrow\ & \text{“}\neg\phi\text{”} \in {}_{\Sigma}CA^{DV}_{CV}(is) \ , \\
\text{“}\phi\text{”}, \text{“}\phi'\text{”} \in {}_{\Sigma}CA^{DV}_{CV}(is) & &\Longrightarrow\ & \begin{cases} \text{“}\phi \wedge \phi'\text{”} \in {}_{\Sigma}CA^{DV}_{CV}(is), \\ \text{“}\phi \vee \phi'\text{”} \in {}_{\Sigma}CA^{DV}_{CV}(is), \\ \text{“}\phi \longrightarrow \phi'\text{”} \in {}_{\Sigma}CA^{DV}_{CV}(is), \\ \text{“}\phi \longleftrightarrow \phi'\text{”} \in {}_{\Sigma}CA^{DV}_{CV}(is). \ , \end{cases} \\
\text{“}\phi\text{”} \in {}_{\Sigma}CA^{DV}_{CV}(is) \wedge x \in DV_s & &\Longrightarrow\ & \begin{cases} \text{“}\forall x\colon \phi\text{”} \in {}_{\Sigma}CA^{DV}_{CV}(is), \\ \text{“}\exists x\colon \phi\text{”} \in {}_{\Sigma}CA^{DV}_{CV}(is) \text{ [for } s \in S]. \ , \end{cases} \\
\text{“}\phi\text{”} \in {}_{\Sigma}CA^{DV}_{CV}(is) \wedge x \in (CV_i)_\omega & &\Longrightarrow\ & \begin{cases} \text{“}\forall x\colon \phi\text{”} \in {}_{\Sigma}CA^{DV}_{CV}(is), \\ \text{“}\exists x\colon \phi\text{”} \in {}_{\Sigma}CA^{DV}_{CV}(is) \text{ [for } i \in I]. \ , \end{cases} \\
v \in (CV_i)_\omega \wedge p \in \mathsf{port}(\mathsf{if}(i)) & &\Longrightarrow\ & \text{“}\widehat{v.p}\text{”} \in {}_{\Sigma}CA^{DV}_{CV}(is) \text{ [for } i \in I] \ , \\
v \in (CV_i)_\omega & &\Longrightarrow\ & \text{“}\wr v \wr\text{”} \in {}_{\Sigma}CA^{DV}_{CV}(is) \text{ [for } i \in I] \ , \\
\left.\begin{aligned} & v \in (CV_i)_\omega \wedge v' \in (CV_j)_\tau \wedge \\ & p \in \mathsf{in}(\mathsf{if}(i)) \wedge p' \in \mathsf{out}(\mathsf{if}(j)) \end{aligned}\right\} & &\Longrightarrow\ & \begin{cases} \text{“}v.p \rightsquigarrow v'.p'\text{”} \in {}_{\Sigma}CA^{DV}_{CV}(is), \\ \text{[for } i, j \in I]. \ . \end{cases}
\end{aligned}
$$

Fig. 64. Inductive definition of architecture assertions.

B.2. Architecture Assertions

B.2.1. Syntax

Definition 17 (Architecture assertions: syntax). The set of all *architecture assertions* over a signature Σ, interface specification $is = (I, \mathsf{if})$ over port specification (PID, tp), data type variables DV, and component variables CV is the smallest set ${}_{\Sigma}CA^{DV}_{CV}(is)$ satisfying the equations in Fig. 64.

Architecture assertions: semantics

$$as \models^{\epsilon,\delta,\kappa}_{A,\iota} \text{“}\mathit{true}\text{”} \quad ,$$

$$\neg(as \models^{\epsilon,\delta,\kappa}_{A,\iota} \text{“}\mathit{false}\text{”}) \quad ,$$

$$as \models^{\epsilon,\delta,\kappa}_{A,\iota} \text{“}b\text{”} \iff A_b \text{ [for } b \in B^0] \ ,$$

$$as \models^{\epsilon,\delta,\kappa}_{A,\iota} \text{“}b(t_1, \cdots, t_n)\text{”} \iff A_b\big({}^{\iota}_{A}[\![\text{“}t_1\text{”}]\!]^{\kappa}_{\mathcal{J}}(as), \cdots, {}^{\iota}_{A}[\![\text{“}t_n\text{”}]\!]^{\kappa}_{\mathcal{J}}(as)\big) \text{ [for } b \in B^{n+1}] \ ,$$

$$as \models^{\epsilon,\delta,\kappa}_{A,\iota} \text{“}t = t' \iff {}^{\iota}_{A}[\![\text{“}t\text{”}]\!]^{\kappa}_{\mathcal{J}}(as) = {}^{\iota}_{A}[\![\text{“}t'\text{”}]\!]^{\kappa}_{\mathcal{J}}(as) \ ,$$

$$as \models^{\epsilon,\delta,\kappa}_{A,\iota} \text{“}\phi \wedge \phi'\text{”} \iff as \models^{\epsilon,\delta,\kappa}_{A,\iota} \text{“}\phi\text{”} \wedge as \models^{\epsilon,\delta,\kappa}_{A,\iota} \text{“}\phi'\text{”} \ ,$$

$$as \models^{\epsilon,\delta,\kappa}_{A,\iota} \text{“}\phi \vee \phi'\text{”} \iff as \models^{\epsilon,\delta,\kappa}_{A,\iota} \text{“}\phi\text{”} \vee as \models^{\epsilon,\delta,\kappa}_{A,\iota} \text{“}\phi'\text{”} \ ,$$

$$as \models^{\epsilon,\delta,\kappa}_{A,\iota} \text{“}\phi \longrightarrow \phi'\text{”} \iff as \models^{\epsilon,\delta,\kappa}_{A,\iota} \text{“}\phi\text{”} \implies as \models^{\epsilon,\delta,\kappa}_{A,\iota} \text{“}\phi'\text{”} \ ,$$

$$as \models^{\epsilon,\delta,\kappa}_{A,\iota} \text{“}\phi \longleftrightarrow \phi'\text{”} \iff as \models^{\epsilon,\delta,\kappa}_{A,\iota} \text{“}\phi\text{”} \iff as \models^{\epsilon,\delta,\kappa}_{A,\iota} \text{“}\phi'\text{”} \ ,$$

$$as \models^{\epsilon,\delta,\kappa}_{A,\iota} \text{“}\exists x\colon \phi\text{”} \iff \begin{cases} \exists x' \in A_s\colon as \models^{\epsilon,\delta,\kappa}_{A,\iota[s\colon x \mapsto x']} \text{“}\phi\text{”} \\ \text{[for } s \in S \text{ and } x \in DV_s] \ , \end{cases}$$

$$as \models^{\epsilon,\delta,\kappa}_{A,\iota} \text{“}\forall x\colon \phi\text{”} \iff \begin{cases} \forall x' \in A_s\colon as \models^{\epsilon,\delta,\kappa}_{A,\iota[s\colon x \mapsto x']} \text{“}\phi\text{”} \\ \text{[for } s \in S \text{ and } x \in DV_s] \ , \end{cases}$$

$$as \models^{\epsilon,\delta,\kappa}_{A,\iota} \text{“}\exists x\colon \phi\text{”} \iff \begin{cases} \exists x' \in \mathcal{C}_{(\epsilon(i),\lambda p\colon\ {}_{A}[\![\omega(p)]\!])}\colon as \models^{\epsilon,\delta,\kappa[i\colon \omega \mapsto \kappa_i[\omega\colon x \mapsto x']]}_{A,\iota} \text{“}\phi\text{”} \\ \text{[for } i \in I \text{ and } x \in (CV_i)_\omega] \ , \end{cases}$$

$$as \models^{\epsilon,\delta,\kappa}_{A,\iota} \text{“}\forall x\colon \phi\text{”} \iff \begin{cases} \forall x' \in \mathcal{C}_{(\epsilon(i),\lambda p\colon\ {}_{A}[\![\omega(p)]\!])}\colon as \models^{\epsilon,\delta,\kappa[i\colon \omega \mapsto \kappa_i[\omega\colon x \mapsto x']]}_{A,\iota} \text{“}\phi\text{”} \\ \text{[for } i \in I \text{ and } x \in (CV_i)_\omega] \ , \end{cases}$$

$$as \models^{\epsilon,\delta,\kappa}_{A,\iota} \text{“}\widehat{v.p}\text{”} \iff \begin{cases} val_{as}\Big(\big((\kappa_i)_\omega(v), \big(\delta(p)\big)\big)\Big) \neq \emptyset \\ \text{[for } i \in I,\ v \in (CV_i)_\omega, \text{ and } p \in \mathsf{port}(\mathsf{if}(i))] \ , \end{cases}$$

$$as \models^{\epsilon,\delta,\kappa}_{A,\iota} \text{“}\wr v \wr\text{”} \iff \begin{cases} |(\kappa_i)_\omega(v)|_{as} \\ \text{[for } i \in I, \text{ and } v \in (CV_i)_\omega] \ , \end{cases}$$

$$as \models^{\epsilon,\delta,\kappa}_{A,\iota} \text{“}v.p \leadsto v'.p'\text{”} \iff \begin{cases} \Big(\big((\kappa_i)_\omega(v'), \delta(p')\big)\Big) \in CN_{as}\Big(\big((\kappa_j)_\tau(v), \delta(p)\big)\Big) \\ \text{[for } i \in I,\ v \in (CV_i)_\omega,\ p \in \mathsf{in}(\mathsf{if}(i)), \\ \ \ j \in I,\ v' \in (CV_j)_\omega,\ p' \in \mathsf{out}(\mathsf{if}(j))] \ . \end{cases}$$

Fig. 65. Recursive definition of satisfaction relation for architecture assertions.

B.2.2. Semantics

Definition 18 (Architecture assertion: semantics). The *semantics* of *architecture assertions* ${}_{\Sigma}CA^{DV}_{CV}(is)$, formulated over interface specification $is = (I, \mathsf{if})$ and port specification (PID, tp), is defined over an algebra $A \in \mathcal{A}(\Sigma)$ with corresponding data type variable assignments $\iota \in \mathcal{I}^{DV}_{A}$, interface interpretation $\epsilon\colon I \to CT_{\mathcal{I}}$, an interpretation $\delta\colon PID \to \mathcal{P}$ for the port identifiers of ps, and component interpretation $\kappa \in \mathcal{K}^{CV}_{\mathcal{C}}$. It is given by a relation $\models^{\epsilon,\delta,\kappa}_{A,\iota} \subseteq AS^{\mathcal{C}}_{\mathcal{T}} \times {}_{\Sigma}CA^{DV}_{CV}(is)$ defined recursively by the equations provided in Fig. 65

Architecture trace assertions: syntax

$$\text{“}\mathit{true}\text{”} \in {}_{\Sigma}CTA^{(DV,CV)}_{(DV',CV')}(is) \ ,$$

$$\text{“}\mathit{false}\text{”} \in {}_{\Sigma}CTA^{(DV,CV)}_{(DV',CV')}(is) \ ,$$

$$\phi \in {}_{\Sigma}CA^{DV \cup DV'}_{CV \cup CV'}(is) \implies \phi \in {}_{\Sigma}CTA^{(DV,CV)}_{(DV',CV')}(is) \ ,$$

$$\text{“}\gamma\text{”} \in {}_{\Sigma}CTA^{(DV,CV)}_{(DV',CV')}(is) \implies \text{“}\neg\gamma\text{”} \in {}_{\Sigma}CTA^{(DV,CV)}_{(DV',CV')}(is) \ ,$$

$$\text{“}\gamma\text{”}, \text{“}\gamma'\text{”} \in {}_{\Sigma}CTA^{(DV,CV)}_{(DV',CV')}(is) \implies \begin{cases} \text{“}\gamma \wedge \gamma'\text{”} \in {}_{\Sigma}CTA^{(DV,CV)}_{(DV',CV')}(is), \\ \text{“}\gamma \vee \gamma'\text{”} \in {}_{\Sigma}CTA^{(DV,CV)}_{(DV',CV')}(is), \\ \text{“}\gamma \longrightarrow \gamma'\text{”} \in {}_{\Sigma}CTA^{(DV,CV)}_{(DV',CV')}(is), \\ \text{“}\gamma \longleftrightarrow \gamma'\text{”} \in {}_{\Sigma}CTA^{(DV,CV)}_{(DV',CV')}(is). \end{cases} ,$$

$$\text{“}\gamma\text{”} \in {}_{\Sigma}CTA^{(DV,CV)}_{(DV',CV')}(is) \implies \text{“}\bigcirc\gamma\text{”},\ \text{“}\Diamond\gamma\text{”},\ \text{“}\Box\gamma\text{”} \in {}_{\Sigma}CTA^{(DV,CV)}_{(DV',CV')}(is) \ ,$$

$$\text{“}\gamma\text{”}, \text{“}\gamma'\text{”} \in {}_{\Sigma}CTA^{(DV,CV)}_{(DV',CV')}(is) \implies \begin{cases} \text{“}\gamma \ \mathcal{U} \ \gamma'\text{”} \in {}_{\Sigma}CTA^{(DV,CV)}_{(DV',CV')}(is), \\ \text{“}\gamma \ \mathcal{W} \ \gamma'\text{”} \in {}_{\Sigma}CTA^{(DV,CV)}_{(DV',CV')}(is). \end{cases} ,$$

$$\left.\begin{matrix} x \in ({DV'}_s)_\omega \quad \wedge \\ \text{“}\gamma\text{”} \in {}_{\Sigma}CTA^{(DV,CV)}_{(DV',CV')}(is) \end{matrix}\right\} \implies \begin{cases} \text{“}\forall x \colon \gamma\text{”} \in {}_{\Sigma}CTA^{(DV,CV)}_{(DV',CV')}(is), \\ \text{“}\exists x \colon \gamma\text{”} \in {}_{\Sigma}CTA^{(DV,CV)}_{(DV',CV')}(is) \ [\text{for } s \in S]. \end{cases} ,$$

$$\left.\begin{matrix} x \in ({CV'}_i)_\omega \quad \wedge \\ \text{“}\gamma\text{”} \in {}_{\Sigma}CTA^{(DV,CV)}_{(DV',CV')}(is) \end{matrix}\right\} \implies \begin{cases} \text{“}\forall x \colon \gamma\text{”} \in {}_{\Sigma}CTA^{(DV,CV)}_{(DV',CV')}(is), \\ \text{“}\exists x \colon \gamma\text{”} \in {}_{\Sigma}CTA^{(DV,CV)}_{(DV',CV')}(is) \ [\text{for } i \in I]. \end{cases} .$$

Fig. 66. Inductive definition of architecture trace assertions.

B.3. Architecture Trace Assertions

B.3.1. Syntax

Definition 19 (Architecture trace assertion: syntax). The set of all *architecture trace assertions* over signature Σ, interface specification $is = (I, \mathsf{if})$ over port specification (PID, tp), data type variables DV, rigid data type variables DV', component variables CV, and rigid component variables CV' is the smallest set ${}_{\Sigma}CTA^{(DV,CV)}_{(DV',CV')}(is)$ satisfying the equations in Fig. 66.

B.3.2. Semantics

Definition 20 (Architecture trace assertion: semantics). The *semantics* of *architecture trace assertions* ${}_{\Sigma}CTA^{(DV,CV)}_{(DV',CV')}(is)$, formulated over interface specification $is = (I, \mathsf{if})$ and port specification (PID, tp), is defined over an algebra $A \in \mathcal{A}(\Sigma)$ with corresponding rigid data type variable assignments $\iota' \in {\mathcal{I}'}^{DV'}_{A}$, interface interpretation $\epsilon \colon I \to CT_{\mathcal{I}}$, an interpretation $\delta \colon PID \to \mathcal{P}$ for the port identifiers of ps, and rigid component interpretation $\kappa' \in {\mathcal{K}'}^{CV}_{\mathcal{C}}$. It is given by a relation $\overset{\epsilon,\delta,\kappa'}{\underset{A,\iota'}{\models}} \subseteq \left((AS^{\mathcal{C}}_{\mathcal{T}})^{\infty} \times \mathbb{N}\right) \times {}_{\Sigma}CTA^{(DV,CV)}_{(DV',CV')}(is)$ defined recursively by the equations provided in Fig. 67

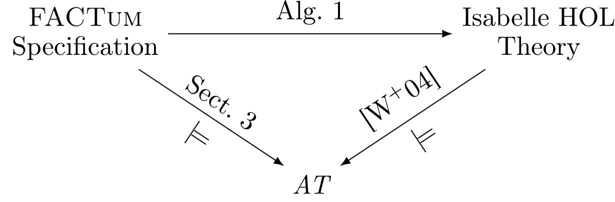
Architecture trace assertions: semantics

$$
\begin{aligned}
&(t,n) \models^{\epsilon,\delta,\kappa'}_{A,\iota'} \text{“}\mathit{true}\text{”} \quad , \\
&\neg\big((t,n) \models^{\epsilon,\delta,\kappa'}_{A,\iota'} \text{“}\mathit{false}\text{”}\big), \\
(t,n) \models^{\epsilon,\delta,\kappa'}_{A,\iota'} \phi \quad &\iff \quad \forall \iota \in \mathcal{I}^{DV}_{A}, \kappa \in \mathcal{K}^{CV}_{\mathcal{C}} : t(n) \models^{\epsilon,\delta,\kappa\cup\kappa'}_{A,\iota\cup\iota'} \phi \ , \\
(t,n) \models^{\epsilon,\delta,\kappa'}_{A,\iota'} \text{“}\gamma \wedge \gamma'\text{”} \quad &\iff \quad (t,n) \models^{\epsilon,\delta,\kappa'}_{A,\iota'} \text{“}\gamma\text{”} \wedge (t,n) \models^{\epsilon,\delta,\kappa'}_{A,\iota'} \text{“}\gamma'\text{”} \ , \\
(t,n) \models^{\epsilon,\delta,\kappa'}_{A,\iota'} \text{“}\gamma \vee \gamma'\text{”} \quad &\iff \quad (t,n) \models^{\epsilon,\delta,\kappa'}_{A,\iota'} \text{“}\gamma\text{”} \vee (t,n) \models^{\epsilon,\delta,\kappa'}_{A,\iota'} \text{“}\gamma'\text{”} \ , \\
(t,n) \models^{\epsilon,\delta,\kappa'}_{A,\iota'} \text{“}\gamma \longrightarrow \gamma'\text{”} \quad &\iff \quad (t,n) \models^{\epsilon,\delta,\kappa'}_{A,\iota'} \text{“}\gamma\text{”} \implies (t,n) \models^{\epsilon,\delta,\kappa'}_{A,\iota'} \text{“}\gamma'\text{”} \ , \\
(t,n) \models^{\epsilon,\delta,\kappa'}_{A,\iota'} \text{“}\gamma \longleftrightarrow \gamma'\text{”} \quad &\iff \quad (t,n) \models^{\epsilon,\delta,\kappa'}_{A,\iota'} \text{“}\gamma\text{”} \iff (t,n) \models^{\epsilon,\delta,\kappa'}_{A,\iota'} \text{“}\gamma'\text{”} \ , \\
(t,n) \models^{\epsilon,\delta,\kappa'}_{A,\iota'} \text{“}\bigcirc\gamma\text{”} \quad &\iff \quad (t,n+1) \models^{\epsilon,\delta,\kappa'}_{A,\iota'} \text{“}\gamma\text{”} \ , \\
(t,n) \models^{\epsilon,\delta,\kappa'}_{A,\iota'} \text{“}\Diamond\gamma\text{”} \quad &\iff \quad \exists n' \geq n \colon (t,n') \models^{\epsilon,\delta,\kappa'}_{A,\iota'} \text{“}\gamma\text{”} \ , \\
(t,n) \models^{\epsilon,\delta,\kappa'}_{A,\iota'} \text{“}\Box\gamma\text{”} \quad &\iff \quad \forall n' \geq n \colon (t,n') \models^{\epsilon,\delta,\kappa'}_{A,\iota'} \text{“}\gamma\text{”} \ , \\
(t,n) \models^{\epsilon,\delta,\kappa'}_{A,\iota'} \text{“}\gamma \ \mathcal{U} \ \gamma'\text{”} \quad &\iff \quad \begin{cases} \exists n' \geq n \colon (t,n') \models^{\epsilon,\delta,\kappa'}_{A,\iota'} \text{“}\gamma'\text{”} \ \wedge \\ \forall n \leq m < n' \colon (t,m) \models^{\epsilon,\delta,\kappa'}_{A,\iota'} \text{“}\gamma\text{”} \ , \end{cases} \\
(t,n) \models^{\epsilon,\delta,\kappa'}_{A,\iota'} \text{“}\gamma \ \mathcal{W} \ \gamma'\text{”} \quad &\iff \quad \begin{cases} (t,n) \models^{\epsilon,\delta,\kappa'}_{A,\iota'} \text{“}\gamma \ \mathcal{U} \ \gamma'\text{”} \ \vee \\ (t,n) \models^{\epsilon,\delta,\kappa'}_{A,\iota'} \text{“}\Box\gamma\text{”} \ , \end{cases} \\
(t,n) \models^{\epsilon,\delta,\kappa'}_{A,\iota'} \text{“}\exists x \colon \gamma\text{”} \quad &\iff \quad \begin{cases} \exists x' \in A_s \colon (t,n) \models^{\epsilon,\delta,\kappa'}_{A,\iota'[s \colon x \mapsto x']} \text{“}\gamma\text{”} \\ [\text{for } s \in S \text{ and } x \in DV'_s] \ , \end{cases} \\
(t,n) \models^{\epsilon,\delta,\kappa'}_{A,\iota'} \text{“}\forall x \colon \gamma\text{”} \quad &\iff \quad \begin{cases} \forall x' \in A_s \colon (t,n) \models^{\epsilon,\delta,\kappa'}_{A,\iota'[s \colon x \mapsto x']} \text{“}\gamma\text{”} \\ [\text{for } s \in S \text{ and } x \in DV'_s] \ , \end{cases} \\
(t,n) \models^{\epsilon,\delta,\kappa'}_{A,\iota'} \text{“}\exists x \colon \gamma\text{”} \quad &\iff \quad \begin{cases} \exists x' \in \mathcal{C}_{(\epsilon(i), \lambda p \colon {}_{A}[\![\omega(p)]\!])} \colon (t,n) \models^{\epsilon,\delta,\kappa'[i \colon \omega \mapsto \kappa'_i[\omega \colon x \mapsto x']]}_{A,\iota'} \text{“}\gamma\text{”} \\ [\text{for } i \in I \text{ and } x \in (CV'_i)_\omega] \ , \end{cases} \\
(t,n) \models^{\epsilon,\delta,\kappa'}_{A,\iota'} \text{“}\forall x \colon \gamma\text{”} \quad &\iff \quad \begin{cases} \forall x' \in \mathcal{C}_{(\epsilon(i), \lambda p \colon {}_{A}[\![\omega(p)]\!])} \colon (t,n) \models^{\epsilon,\delta,\kappa'[i \colon \omega \mapsto \kappa'_i[\omega \colon x \mapsto x']]}_{A,\iota'} \text{“}\gamma\text{”} \\ [\text{for } i \in I \text{ and } x \in (CV'_i)_\omega] \ . \end{cases}
\end{aligned}
$$

Fig. 67. Recursive definition of satisfaction relation for architecture trace assertions.

C. Soundness of Algorithm 1

In the following, we provide an argument of why Alg. 1 preserves the semantics of a FACTUM specification. The following diagram provides an overview of our reasoning:

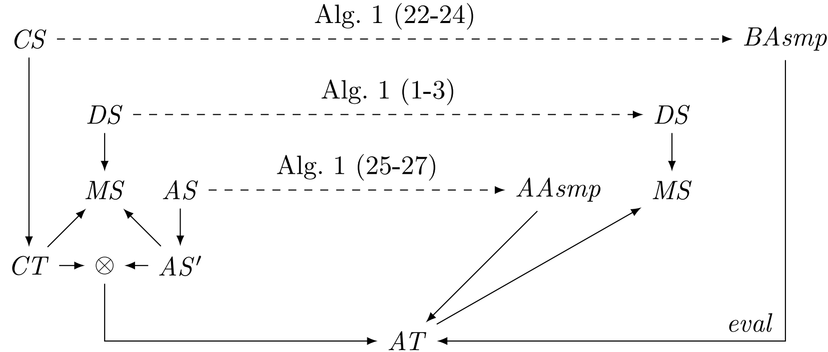


We need to show that a set of architecture traces AT satisfies a FACTUM specification iff it satisfies the Isabelle/HOL theory generated from the specification by algorithm 1.

To this end, we assume the existence of a FACTUM specification $PS = (DS, CS, AS)$, consisting of an algebraic specification of datatypes DS, a specification of component types CT, and an architecture specification AS.

C.1. Case $\Longrightarrow$

We fix a set of architecture traces AT and assume that it satisfies PS. We show that AT also satisfies the Isabelle theory generated from PS by algorithm 1. To this end, we fix an architecture trace $t \in AT$ and show that t satisfies the corresponding Isabelle theory. Again, the idea of the argument is depicted by the following diagram:



Since AT satisfies PS, the semantics of FACTUM (discussed in Sect. 3) requires the existence of an architecture specification AS' which satisfies AS, such that $t \in AS'$. Thus, t also satisfies the locale assumptions generated from AS by lines $25 - 27$ of algorithm 1.

Similarly, since AT satisfies PS, the semantics of FACTUM requires the existence of a behavior CT_c, for each component c, which satisfies the corresponding behavior specification CS_c. Moreover, the semantics of FACTUM also requires the existence of a behavior trace t', such that $\Pi_c(t)^\frown t' \in CT_c$. Thus, according to the definition of $eval$ (discussed in Sect. 4.1.5) we have $eval(c, t, t', \gamma)$ for each component c and locale assumption γ (generated by lines $25 - 27$ of Alg. 1).

Thus, t' fulfills all locale assumptions generated by Alg. 1 and thus it satisfies the the Isabelle theory generated from PS.

C.2. Case $\Longleftarrow$

We may use a symmetric argument as the one presented in case C.1 for the reverse direction.