



Hierarchical Specification and Verification of Architectural Design Patterns

Diego Marmosoler^(✉) 

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

Abstract. Architectural design patterns capture architectural design experience and provide abstract solutions to recurring architectural design problems. Their description is usually expressed informally and it is not verified whether the proposed specification indeed solves the original design problem. As a consequence, an architect cannot fully rely on the specification when implementing a pattern to solve a certain problem. To address this issue, we propose an approach for the specification and verification of architectural design patterns. Our approach is based on interactive theorem proving and leverages the hierarchical nature of patterns to foster reuse of verification results. The following paper presents FACTum, a methodology and corresponding specification techniques to support the formal specification of patterns. Moreover, it describes an algorithm to map a given FACTum specification to a corresponding Isabelle/HOL theory and shows its soundness. Finally, the paper demonstrates the approach by verifying versions of three widely used patterns: the singleton, the publisher-subscriber, and the black-board pattern.

Keywords: Architectural design patterns
Interactive theorem proving · Dynamic architectures
Algebraic specification · Configuration traces

1 Introduction

Architectural design patterns capture architectural design experience and provide abstract solutions to recurring architectural design problems. They are an important concept in software engineering and regarded as one of the major tools to support an architect in the conceptualization and analysis of software systems [1]. The importance of patterns resulted in a panoply of pattern descriptions in literature [1–3]. They usually consist of a description of some key architectural constraints imposed by the pattern, such as involved data types, types of components, and assertions about the activation/deactivation of components as well as connections between component ports. These descriptions are usually highly informal and the claim that they indeed solve a certain design problem remains unverified. As a consequence, an architect cannot fully rely on a pattern’s specification to solve a design problem faced during the development of a

© The Author(s) 2018

A. Russo and A. Schürr (Eds.): FASE 2018, LNCS 10802, pp. 149–168, 2018.

https://doi.org/10.1007/978-3-319-89363-1_9

new architecture. Moreover, verified pattern descriptions are a necessary precondition for automatic pattern conformance analyses, since missing assertions in a pattern’s specification renders their detection impossible. Compared to concrete architectures, architectural design patterns pose several new challenges to the specification as well as the verification:

- *C1: Axiomatic Specifications.* Compared to traditional architectural specifications, specifications of patterns are usually axiomatic, focusing on a few, but important properties.
- *C2: Dynamic Aspects:* Pattern specifications usually involve the specification of dynamic aspects, such as instantiation of components and reconfiguration of connections.
- *C3: Hierarchical Specifications:* Pattern specifications usually build on each other, i.e., the specification of a pattern may instantiate the specification of another pattern.

This is why traditional techniques for the specification and verification of concrete architectures are not well-suited to be applied for the specification and verification of patterns.

Therefore, we propose an approach for the formal specification and verification of architectural design patterns which is based on interactive theorem proving [4]. Our approach is built on top of a pre-existing model of dynamic architectures [5,6] and its formalization in Isabelle/HOL [7] which comes with a calculus to support reasoning about such architectures [8]. Our approach provides techniques to specify patterns and corresponding design problems and allows to map a specification to a corresponding Isabelle/HOL theory [9]. The theory and the corresponding calculus can then be used to verify that a specification indeed solves the design problem the pattern claims to solve.

With this paper, we elaborate on our previous work by providing the following contributions: First, we present FACTum, a novel approach for the formal specification of architecture design patterns. Second, we provide an improved version of the algorithm to map a given FACTum specification to a corresponding Isabelle/HOL theory and show soundness of the mapping. Third, we demonstrate the approach by specifying and verifying versions of three architectural design patterns: the singleton pattern, the publisher subscriber pattern, and the blackboard pattern.

The remainder of the paper is structured as follows: In Sect. 2, we provide necessary background on interactive theorem proving and configuration traces (our model of dynamic architectures). We then describe our approach to specify patterns in Sect. 3. To this end, we define the notion of (hierarchical) pattern specification and demonstrate it by specifying three architectural design patterns. In Sect. 4, we first define the semantics of a pattern specification in terms of configuration traces. Then, we provide an algorithm to map a given specification to a corresponding Isabelle/HOL theory and show its soundness, i.e., that the semantics of a specification is indeed preserved by the algorithm. We proceed with an overview of related work in Sect. 5 and conclude the paper in Sect. 6 with a brief discussion about how the approach addresses the challenges C1–C3 identified above.

2 Background

In the following, we provide some background on which our work is build.

2.1 Interactive Theorem Proving

Interactive theorem proving (ITP) is a semi-automatic approach for the development of formal theories. Therefore, a set of proof assistants [4] have been developed to support a human in the development of formal proofs. Since our approach is based on Isabelle/HOL [9], in the following we describe some relevant features about this specific prover.

In general, Isabelle is an LCF-style [10] theorem prover based on Standard ML. It provides a so-called meta-logic on which different object logics are based. Isabelle/HOL is one of them, implementing higher-order logic for Isabelle. It integrates a prover IDE and comes with an extensive library of theories from various domains. New theories are then developed by defining terms of a certain type and deriving theorems from these definitions. Data types can be specified in Isabelle/HOL in terms of freely generated, inductive data type definitions [11]. Axiomatic specification of data types is also supported in terms of type classes [12]. To support the specification of theories over the data types, Isabelle/HOL provides tools for inductive definitions and recursive function definitions. Moreover, Isabelle/HOL provides a structured proof language called Isabelle/Isar [13] and a set of logical reasoners to support the verification of theorems. Modularization of theories is achieved through the notion of locales [14] in which an interface is specified in terms of sets of functions (called parameters) with corresponding assumptions about their behavior. Locales can extend other locales and may be instantiated by concrete definitions of the corresponding parameters.

2.2 A Model of Dynamic Architectures

Since architectures implementing an ADP may be dynamic as well (in the sense that components of a certain type can be instantiated over time), our approach is based on a model of dynamic architectures. One way to model such architectures is in terms of *sets* of configuration traces [5,6], i.e., streams [15,16] over architecture configurations. Thereby, architecture configurations can be thought of as snapshots of the architecture during execution. Thus, they consist of a set of (active) components with their ports valued by messages and connections between the ports of the components. Moreover, components of a certain type may be parametrized by a set of messages.

Example 1 (Configuration trace). Assuming that A, \dots, Z and $1, \dots, 9$ are messages. Figure 1 depicts a configuration trace t with corresponding architecture configurations $t(0) = k_0$, $t(1) = k_1$, and $t(2) = k_2$. Architecture configuration k_1 , for example, consists of two active components named c_1 and c_2 . Thereby, component c_1 is parametrized by $\{A\}$, has one input port i_0 valued with $\{8\}$, and three output ports o_0, o_1, o_2 , valued with $\{1\}$, $\{G\}$, and $\{7\}$. \square

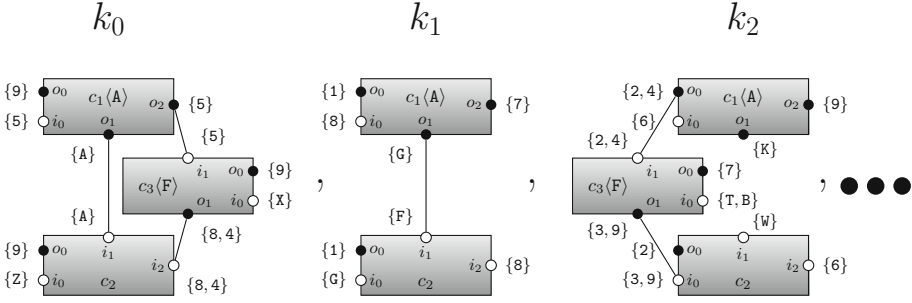


Fig. 1. Configuration trace with its first three architecture configurations.

Note that the model allows components to be valued by a set of messages, rather than just a single message, at each point in time. To evaluate the behavior of a single component, the model comes with an operator $\Pi_c(t)$ to extract the behavior of a single component c out of a given configuration trace t .

The model of configuration traces is also implemented by a corresponding Isabelle/HOL theory which is available through the archive of formal proofs [7]. The implementation formalizes a configuration trace as a function $trace = nat \rightarrow cnf$ and provides an interface to the model in terms of a locale “dynamic.component”. The locale can be instantiated with components of a dynamic architecture by providing definitions for two parameters:

- $tCMP: id \times cnf \rightarrow cmp$: an operator to obtain a component cmp with a certain identifier id from an architecture configuration cnf , and
- $active: id \times cnf \rightarrow bool$: a predicate to assert whether a certain component with identifier id is activated within an architecture configuration cnf .

For each dynamic component instantiating the locale, a set of definitions is provided to support the specification of its behavior [17]. Moreover, a calculus to reason about the behavior of the component in a dynamic context is provided [8].

3 Specifying Architectural Design Patterns

In the following, we describe FACTum, an approach to specify architectural design patterns. Therefore, we first provide a definition of the different parts of a pattern specification and then we explain each part in more detail. We conclude the section with an exemplary specification of three patterns: the singleton, the publisher subscriber, and the blackboard pattern. Thereby, 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.

Definition 1 (Pattern specification). A pattern specification is a 5-tuple (VAR, DS, IS, CT, AS) , consisting of:

- Variables $VAR = (V, V', C, C')$ with
 - data type variables V and so-called rigid data type variables V' (variables with a fixed interpretation during execution) and
 - component variables C and rigid component variables C' .
- A datatype specification $DS = (\Sigma, DA, Gen)$ with
 - a signature $\Sigma = (S, F, B)$, containing sorts S and function/predicate symbols F/B for a pattern's data types,
 - a set of data type assertions DA specifying the meaning of the signature symbols in terms of a set of axioms, and
 - a set of generator clauses Gen to construct data types.
- An interface specification $IS = (P, tp, IF)$ with
 - a set of ports P and corresponding type function $tp: P \rightarrow S$ which assigns a sort to each port,
 - a set of interfaces $(CP, IP, OP) \in IF$ with input ports $IP \subseteq P$ and output ports $OP \subseteq P$, as well as a set of configuration parameters $CP \subseteq P$.
- A component type specification $(CT_{if})_{if \in IF}$ which assigns assertions CT_{if} about the behavior of a component to each interface $if \in IF$.
- A set of architectural assertions AS , which specify activation and deactivation of components and connections between the component's ports.

Since a pattern specification may also instantiate other pattern specifications, we require that for each instantiated pattern $(VAR', DS', IS', CT', AS')$, the specification contains an additional *port instantiation* $(\eta_{i'})_{i' \in IF'}$, with *injective* functions $\eta_{i'}: CP' \cup IP' \cup OP' \rightarrow CP \cup IP \cup OP$, such that $\eta_{i'}(CP') \subseteq CP$, $\eta_{i'}(IP') \subseteq IP$, and $\eta_{i'}(OP') \subseteq OP$, for some $(CP, IP, OP) \in IF$. Thereby, we require that for each $(CP', IP', OP') \in IF'$ and $p' \in CP' \cup IP' \cup OP'$ the corresponding data type refines the type of p' , i.e., that $tp(\eta_{i'}(p'))$ refines $(tp'(p'))$.

In the following, we explain the different parts of a FACTum specification in more detail.

3.1 Specifying Data Types

The data types involved in a pattern specification can be specified using *algebraic specification techniques* [18, 19]. Algebraic specifications usually consist of two parts: First, a signature $\Sigma = (S, F, B)$, specifying a set of sorts S and function/predicate symbols F/B , typed by a list of sorts. In addition, an algebraic specification provides a set of axioms DA to assign meaning to the symbols of Σ . These axioms specify the characteristic properties of the data types used by a pattern specification and are formulated over the symbols of F and B , respectively. Finally, a data type specification may require that all elements of the corresponding type are constructed by corresponding constructor terms Gen , i.e., that each element of the corresponding type is build up from symbols of Gen .

3.2 Specifying Interfaces

The specification of interfaces proceeds then in two steps: First, ports are specified by providing a set of ports P and a corresponding mapping $tp: P \rightarrow S$ to specify which types of data may be exchanged through each port. Then, a set of interfaces (CP, IP, OP) is specified by declaring input ports $IP \subseteq P$, output ports $OP \subseteq P$, and a set of configuration parameters $CP \subseteq P$. Thereby, configuration parameters are a way to parametrize components of a certain type and they can be thought of as ports with a predefined value which is fixed for each component.

Interfaces can then be specified using so-called *configuration diagrams* consisting of a graphical depiction of the involved interfaces (see Sect. 3.6 for examples). Thereby, each interface consists of two parts: A name followed by a list of configuration parameters (enclosed between ‘ \langle ’ and ‘ \rangle ’). Input and output ports are represented by empty and filled circles, respectively.

3.3 Specifying Component Types

Component types are specified by assigning assertions about the input/output behavior to the interfaces. Thereby, configuration parameters can be used to distinguish between different components of a certain type.

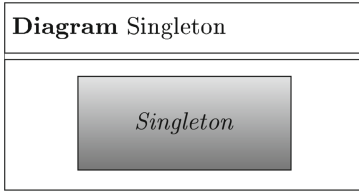
The assertions are expressed in terms of linear temporal logic equations [20] formulated over the signature Σ by using port names as free variables. For example, the term “ $\square(c.p = \text{POS} \longrightarrow c.o \geq 1)$ ” denotes an assertion that port o of component c , for which configuration parameter p has the value POS (for positive), is guaranteed to be greater or equal to 1 for the whole execution of the system.

3.4 Specifying Activation and Connection Assertions

Finally, a set of assertions about the activation and deactivation of components as well as assertions about connections between component ports are specified. Both types of assertions may be expressed in terms of so called *configuration trace assertions*, i.e., linear temporal logic formulæ with special predicates to denote activation of components and port connections. Thereby, $c.p$ denotes the valuation of port p of a component c (where $\widehat{c.p}$ denotes that port p of component c is valuated, at all), $\|c\|$ denotes that a component c is currently active, and $c.p \rightsquigarrow c'.p'$ denotes that output port p of component c is connected to input port p' of component c' .

3.5 Specifying Pattern Instantiations

As described above, pattern specifications may be built on top of other pattern specifications by instantiating their component types. Such instantiations can be directly specified in a pattern’s configuration diagram by annotating the

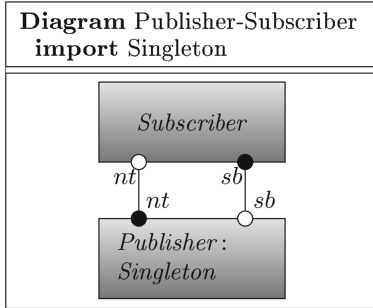


(a) Configuration diagram.

ASpec Singleton	for Singleton
var $c:$	<i>Singleton</i>
rig $c':$	<i>Singleton</i>
$\square(\exists c: \ c\)$	(1)
$\exists c': (\square(\forall c: (\ c\ \rightarrow c = c')))$	(2)

(b) Activation specification.

Fig. 2. Specification of the singleton pattern.



(a) Configuration diagram.

DTSpec subscription(id,evt)
generated by sub id evt, unsub id evt

(b) Data type specification.

PSpec Publisher-Subscriber
$sb:$ <i>subscription</i>(id, \wp(evt))
$nt:$ evt \times msg

(c) Port specification.

Fig. 3. Specification of the publisher subscriber pattern.

corresponding interfaces. To denote that a certain component type t of the specification is an instance of component type t' (from the instantiated pattern), we simply write $t : t'$ followed by a corresponding *port mapping* $[p'_i, p'_o \mapsto p_i, p_o]$, which assigns a port of t to each port of t' .

3.6 Example: An Initial Pattern Hierarchy

In the following, we demonstrate the FACTum approach by specifying variants of three well-known patterns: the singleton pattern, the publisher subscriber pattern, and the blackboard pattern. Thereby, the publisher component of the publisher subscriber pattern is modeled as an instance of the singleton, whereas the blackboard pattern is specified by instantiating the publisher subscriber pattern.

Singleton. The singleton pattern is a pattern for dynamic architectures in which, for a certain type of component, it is desired to have only one active instance at all points in time. Figure 2 depicts a possible specification of the pattern in terms of a configuration diagram and a corresponding activation specification. Since the pattern is only concerned with activation of components, we do neither have data types, nor port specifications for that pattern.

Interfaces. The interface is specified by the configuration diagram in Fig. 2a: It consists of a single interface *Singleton* and does not require any special ports.

Architectural Assertions. Activation assertions are formalized by the specification depicted in Fig. 2b: With Eq. 1 we require that there exists a component c which is always activated and with Eq. 2 we require the component to be unique. In our version of the singleton, we require that the singleton component is not allowed to change over time. This is why variable c is declared to be rigid in Fig. 2b. Indeed, other versions of the singleton are possible in which the singleton may change over time.

Publisher Subscriber. We now proceed by specifying a version of the publisher subscriber pattern. Such patterns are used for architectures in which so-called subscriber components can subscribe for certain messages from other, so-called publisher components. Figure 3 depicts a possible specification of the pattern in terms of a data type specification, port specification, and corresponding configuration diagram.

Data Types. In a publisher subscriber pattern we usually have two types of messages: subscriptions and unsubscriptions. Figure 3b 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 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 ”.

Ports. Two port types are specified over these data types by the specification given in Fig. 3c: a type sb which allows to exchange subscriptions to a specific event and type nt which allows to exchange messages associated to any event.

Interfaces. The configuration diagram depicted in Fig. 3a depicts the specification of the interfaces of the two types of components: 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 *Subscriber* is defined with an input port nt receiving notifications and an output port sb to send out subscriptions. As stated in the beginning, we want a publisher to be unique and activated which is why it is specified as *Publisher:Singleton*, meaning that it is considered to be an instance of the *Singleton* type of the specification of the singleton pattern.

Architectural Assertions. Activation assertions for publisher subscriber architectures are mainly inherited from the singleton pattern: since a publisher is specified to be a singleton, a publisher component is unique and always activated. Moreover, two connection assertions for publisher subscriber architectures are specified in Fig. 4: Eq. (3) requires a publisher’s input port sb to be connected to the corresponding output port of every active subscriber which sends some

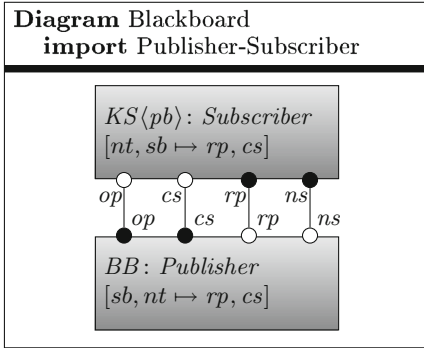
ASpec Publisher-Subscriber	for Publisher-Subscriber
var $s:$ $p:$ $m:$ $E:$ rig $s':$ $e:$	<i>Subscriber</i> <i>Publisher</i> msg $\varphi(\text{evt})$ <i>Subscriber</i> evt
$\square \left(\ p\ \wedge \ s\ \wedge \widehat{s.sb} \longrightarrow p.sb \rightsquigarrow s.sb \right)$ (3)	
$\square \left(\ s'\ \wedge (\exists E: \text{sub } s' E \in s'.sb \wedge e \in E) \right)$ (4)	
$\longrightarrow \left((\ p\ \wedge \ s'\ \wedge (e,m) \in p.nt \longrightarrow s'.nt \rightsquigarrow p.nt) \mathcal{W} (\ s'\ \wedge (\exists E: \text{unsub } s' E \in s'.sb \wedge e \in E)) \right)$	

Fig. 4. Architectural constraints for the blackboard pattern.

message. Equation (4), on the other hand, requires a subscriber's input port nt to be connected to the corresponding output port of the publisher, whenever the latter sends a message for which the subscriber is subscribed.

Blackboard. We conclude our example by specifying a dynamic version of the blackboard pattern. A blackboard architecture is usually used for the task of collaborative problem solving, i.e., a set of components work together to solve an overall, complex problem. Our specification of the pattern is depicted in Fig. 5 and consists of a data type specification, port specification, and corresponding configuration diagram.

Data Types. Blackboard architectures usually work with *problems* and *solutions* for them. Figure 5b 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 to be solved first. Therefore, we assume the existence of a *subproblem relation* $\prec \subseteq \text{PROB} \times \text{PROB}$. 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 (Eq. (5)) 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, without loss of generality, that the final solution for a problem is always unique. Thus, we assume the existence of a function $\text{solve}: \text{PROB} \rightarrow \text{SOL}$ 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 calculate this function.



(a) Configuration diagram.

DTSpec ProbSol	imports SET
\prec :	$PROB \times PROB$
<i>solve</i> :	$PROB \rightarrow SOL$
<i>well-founded</i> (\prec)	(5)

(b) Data type specification.

PSpec BPort	
<i>rp</i> :	$PROB \times \wp(PROB)$
<i>ns, cs</i> :	$PROB \times SOL$
<i>op, prob</i> :	$PROB$

(c) Port specification.

Fig. 5. Specification of the blackboard pattern.

Ports. In Fig. 5c, we specify 4 ports for the pattern:

- rp is used to exchange a problem $p \in PROB$ which a knowledge source is able to solve, together with a set of subproblems $P \subseteq PROB$ the knowledge source requires to be solved first.
- ns is used to exchange a problem $p \in PROB$ solved by a knowledge source, together with the corresponding solution $s \in SOL$.
- op is used to exchange a set $P \subseteq PROB$ of all the problems which still need to be solved.
- cs is used to exchange solutions $s \in SOL$ for problems $p \in PROB$.

Moreover a configuration parameter $prob$ is specified to parametrize knowledge source according to the problems $p \in PROB$ they can solve.

Interfaces. A blackboard pattern usually involves two types of components: blackboards and knowledge sources. The corresponding interfaces are specified by the configuration diagram in Fig. 5a. Since our version of the blackboard pattern is specified to be an instance of the publisher subscriber pattern, we import the corresponding pattern specification in the header of the diagram. We then specify two interfaces. The blackboard interface is denoted BB and is declared to be an instance of a *Publisher* component in a publisher subscriber pattern. It consists of two input ports rp and ns to receive required subproblems and new solutions. Moreover, it specifies two output ports op and cs to communicate currently open problems and solutions for all currently solved problems. Thereby, port rp is specified to be an instance of port sb of a publisher and port cs to be an instance of a publisher’s nt port.

The interface for knowledge sources is denoted KS and is declared to be an instance of a *Subscriber* component in a publisher subscriber pattern. Note that each knowledge source can only solve certain problems, which is why a knowledge source is parameterized by a problem “ $prob$ ”. The specification of ports actually mirrors the corresponding specification of the blackboard interface. Thus, a knowledge source is required to have two input ports op and cs to

BSpec Blackboard		for <i>BB</i> of Blackboard
var	$p:$	PROB
	$P:$	PROB SET
rig	$p':$	PROB
	$s':$	SOL
<hr style="border-top: 1px dashed black;"/>		
	$\square((p', s') \in ns \longrightarrow \diamond((p', s') \in cs))$	(6)
	$\square((p, P) \in rp \longrightarrow (\forall p' \in P: (\diamond(p' \in op))))$	(7)
	$\square(p' \in op \longrightarrow (p' \in op \mathcal{W} (p', solve(p')) \in cs)$	(8)

Fig. 6. Specification of behavior for blackboard components.

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. Thereby, port rp is specified to be an instance of a subscribers nt port and port cs to be an instance of a subscribers sb port, respectively.

Component Types. A blackboard provides the *current state* towards solving the original problem and forwards problems and solutions from knowledge sources. Figure 6 provides a specification of the blackboard's behavior in terms of three behavior 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. 6).
- If, on its input port rp , it gets notified that solutions for some subproblems P are required in order to solve a certain problem p , these problems are eventually provided at its output port op (Eq. (7)).
- A problem p' is provided at its output port op as long as it is not solved (Eq. (8)).

Note that the last assertion (Eq. (8)) is formulated using a *weak* until operator which is defined as follows: $\gamma' \mathcal{W} \gamma \stackrel{\text{def}}{=} \square(\gamma') \vee (\gamma' \mathcal{U} \gamma)$.

A knowledge source receives open problems via op and provides solutions for other problems via cs . It might contribute to the solution of the original problem by solving currently open subproblems. Figure 7 provides a specification of the knowledge sources's behavior in terms of four behavior assertions:

- If a knowledge source (able to solve a problem pp) requires some subproblems P to be solved in order to solve pp and it gets solutions for all these subproblems p' on its input port cs , then it eventually solves pp and provides the solution on its output port ns (Eq. (9)).
- To solve a problem pp , a knowledge source requires solutions only for smaller problems $p \in P$ (Eq. (10)).
- A knowledge source will eventually communicate its ability to solve an open problem pp via its output port rp (Eq. (11)).
- A knowledge source does not unsubscribe from receiving solutions for subproblems it required until it indeed received these solutions (Eq. (12)).

BSpec Knowledge Source		for $ks = KS\langle pp \rangle$ of Blackboard
var	$p:$	PROB
	$P:$	$\wp(\text{PROB})$
rig	$p':$	PROB
\square	$(\forall (pp, P) \in rp: ((\forall p' \in P: \diamond(p', solve(p')) \in cs) \longrightarrow \diamond(pp, solve(pp)) \in ns))$	(9)
\square	$(\forall (pp, P) \in rp: \forall p \in P: p \prec pp)$	(10)
\square	$(pp \in op \longrightarrow \diamond(\exists P: (pp, P) \in rp))$	(11)
\square	$(\text{sub } ks P = rp \longrightarrow (\neg \exists P': p \in P' \wedge \text{unsub } ks P' = rp \ \mathcal{W}(p, solvep) \in cs))$	(12)

Fig. 7. Specification of behavior for knowledge source components.

ASpec Blackboard		for Blackboard
var	$ks:$	$KS\langle pp \rangle$
	$bb:$	BB
rig	$ks':$	$KS\langle pp \rangle$
\square	$(\ ks'\ \wedge pp \in ks'.op \longrightarrow (\ ks'\ \ \mathcal{W} \ \ ks'\ \wedge (pp, solve(pp)) \in ks'.ns))$	(13)
\square	$(\ ks\ \wedge \ bb\ \wedge \widehat{bb.op} \longrightarrow ks.op \rightsquigarrow bb.op)$	(14)
\square	$(\ bb\ \wedge \ ks\ \wedge \widehat{ks.ns} \longrightarrow bb.ns \rightsquigarrow ks.ns)$	(15)

Fig. 8. Specification of activation constraints for blackboard architectures.

Architectural Assertions. Activation constraint for blackboards are mainly inherited from the singleton pattern: since a blackboard is specified to be an instance of a publisher which is again an instance of a singleton, a blackboard component is unique and always activated. Activation constraint for knowledge sources are provided in Fig. 8 by Eq. (13): Whenever a knowledge source (able to solve a problem pp) gets notified about a request to solve pp , it stays active until pp is indeed solved. Connection assertions for the blackboard pattern are mainly inherited from the corresponding specification of the publisher subscriber pattern (for ports rp and cs , respectively). Two additional assertions, however, are provided in Fig. 8: with Eq. 14 we require input ports op of active blackboard components to be connected to the corresponding output ports of knowledge sources and with Eq. 15 we require a similar property for port ns .

4 Verifying Architectural Design Patterns

In the last section we presented FACTum, a methodology and corresponding techniques to specify architectural design pattern. Thereby, we relied on an intuitive understanding of the semantics of the techniques. In the following, we first provide a more formal definition of the semantics of a FACTum specification. Then, we describe an algorithm to map a given specification to a corresponding Isabelle/HOL theory and we show soundness of the algorithm.

4.1 Semantics of Pattern Specifications

The semantics of a pattern specification is given in terms of sets of configuration traces introduced in Sect. 2.

Definition 2 (Semantics of Pattern Specification). *The semantics of a pattern specification (VAR, DS, IS, CT, AS) is given by a 5-tuple $(\mathcal{A}, \mathcal{P}, \mathcal{T}, \mathcal{C}, AT)$, consisting of:*

- an algebra $\mathcal{A} = ((A_s)_{s \in S}, (f^A)_{f \in F}, (p^A)_{p \in B})$ for Σ ,
- a set of ports \mathcal{P} with cardinality greater or equal to the cardinality of P ,
- port typing $\mathcal{T}: \mathcal{P} \rightarrow \wp(\mathcal{M})$ with $\mathcal{M} = \bigcup_{s \in S} (A_s)$,
- a nonempty set of component identifiers \mathcal{C}_{if} for each component interface $if \in CI_{\mathcal{T}}^{\mathcal{P}}$, and
- an architecture $AT \in DA_{\mathcal{T}}^{\mathcal{C}}$;

such that for all port interpretations $\delta: P \rightarrow \mathcal{P}$ (injective mappings which respect tp and \mathcal{T}), variable interpretations $\iota: V \rightarrow A$ and $\iota': V' \rightarrow A$, and component variable interpretations $\kappa: C \rightarrow \mathcal{C}$ and $\kappa': C' \rightarrow \mathcal{C}$ (respecting interface types) the following conditions hold:

- A is an algebra for the data type specification: $A, \iota \models DS$,
- the projection to the behavior of a component c for every configuration trace t of the architecture satisfies the corresponding behavior specification: $\forall c \in C_{\mathcal{T}}^{\mathcal{C}}, t \in AT: \Pi_c(t) \hat{=} CT_c$, and
- all configuration traces t of the architecture satisfy the architectural assertions: $\forall t \in AT: t, \iota', \kappa' \models AS$.

4.2 Mapping to Isabelle/HOL

Algorithm 1 describes how to systematically transfer a pattern specification to a corresponding Isabelle/HOL theory. In general, the transformation is done in 4 main steps: (i) The specified data types are transferred to corresponding Isabelle/HOL data type specifications (ii) An Isabelle locale is created for the corresponding pattern which imports other locales for each instantiated pattern. (iii) Specifications of component behavior are added as assumptions. (iv) Activation and connection assertions are provided as assumptions.

The following soundness criterion guarantees that Algorithm 1 indeed preserves the semantics of a pattern specification.

Theorem 1 (Soundness of Algorithm 1). *For every pattern specification PT , and model T of the Isabelle/HOL locale (as specified in [21]) generated by Algorithm 1, there exists a T' such that $T' \models PT$ (as defined by Definition 2) and T' is isomorphic to T ; and vice versa.*

Note that the generated theory is based on Isabelle/HOLs implementation of configuration traces [7]. Thus, a calculus is instantiated for each component type which provides a set of rules to reason about the specification of the behavior of components of that type.

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

Input: (VAR, DS, IS, CT, AS) {pattern specification according to Definition 1}
Output: An Isabelle/HOL theory for the specification

- 1: create Isabelle/HOL data type specification for DS
- 2: create Isabelle/HOL locale for the pattern
- 3: **for all** Interfaces $i = (CP, IP, OP) \in IF$ **do**
- 4: **if** i instantiates a component of another pattern **then**
- 5: import the corresponding locale
- 6: create instance of ports according to δ_i
- 7: **else**
- 8: import locale “dynamic_component” of theory “Configuration_Traces” [8]
- 9: **end if**
- 10: create instance of locale parameters $tCMP$ and $active$
- 11: **for all** configuration parameters $p \in CP$ which are not instances **do**
- 12: create locale parameter p of type $tp(p)$
- 13: create locale assumption “ $\forall x. \exists c. x = p(c)$ ”
- 14: **end for**
- 15: **for all** ports $p \in IP \cup OP$ which are not instances **do**
- 16: create locale parameter p of type $tp(p)$
- 17: **end for**
- 18: **for all** behavior assertions $b \in CT_i$ **do**
- 19: create locale assumption for b using def. of theory “Configuration_Traces” [8]
- 20: **end for**
- 21: **end for**
- 22: **for all** activation/connection assertions $c \in AS$ **do**
- 23: create locale assertion for c
- 24: **end for**

4.3 Example: Pattern Hierarchy

Algorithm 1 can be used to transfer a given pattern specification to a corresponding Isabelle/HOL theory where it is subject to formal verification. This is demonstrated by applying it to the specification of the singleton, publisher subscriber, and blackboard pattern presented in Sect. 3.6. To demonstrate the verification capabilities, we then proof one characteristic property for each pattern. The corresponding Isabelle/HOL theory files are provided online [22].

Singleton. We first come up with a basic property for singleton components which ensures that there exists indeed a *unique* component of the corresponding type which is always activated:

$$\exists!c: \Box (\|c\|). \quad (16)$$

Publisher Subscriber. Lets now turn to the publisher subscriber pattern. First of all, remember that the publisher component was specified to be an instance of the singleton pattern which is why all results from the verification of the singleton pattern are lifted to the publisher component. Thus, we get

an equivalent result as Eq. (16) for free. Moreover, we can use the additional assertions imposed by the specification to come up with another property for the publisher subscriber pattern which guarantees that a subscriber indeed receives all the messages for which he is subscribed:

$$\square \left(\|c\| \wedge \text{sub } c \ E \in c.sb \longrightarrow \right. \quad (17)$$

$$\left. ((e, m) \in p.nt \wedge e \in E \longrightarrow (e, m) \in c.sb) \ \mathcal{W} \ (\text{unsub } c \ E' \in c.sb \wedge e \in E') \right).$$

Note that the proof of the above property is based on Eq. (16) inherited from the singleton pattern. Indeed, the hierarchical nature of FACTum allows for reuse of verification results from instantiated patterns.

Blackboard. Again, the properties verified for singletons (Eq. (16)) as well as the properties verified for publisher subscriber architectures (Eq. (17)) are inherited for the blackboard specification. In the following, we use these properties to verify another property for blackboard architectures: A blackboard pattern guarantees that if for each open (sub-)problem, there exists a knowledge source which is able to solve the corresponding problem:

$$\square \left(\forall p' \in bb'.op : \diamond (\|ks_{p'}\|) \right), \quad (18)$$

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

$$\square \left(p' \in bb'.rp \longrightarrow \diamond (p', \text{solve}(p')) \in bb'.cs \right). \quad (19)$$

5 Related Work

Related work can be found in three different areas.

Formal Specification of Architectural Styles. Over the last years, several approaches emerged to support the formal specification of architectural design patterns. One of the first attempts in this direction was Wright [23] which provided the possibility to specify architectural styles which is similar to our notion of architectural design pattern. More recent approaches to specify styles are based on the BIP framework [24] and provide logics [25] as well as graphical notation [26] to specify styles. There are, however, two differences of these approaches to the work presented in this paper: One difference concerns the expressive power of the specification techniques. While the above approaches focus mainly on the specification of patterns for static architectures, we allow for the specification of static as well as dynamic architectures. Another difference arises from the scope of the work. While the above approaches focus mainly on the specification of patterns, our focus is more on the verification of such specifications.

Verification of Architectural Styles and Patterns. Recently, some approaches emerged which focus on the verification of architectural styles and patterns. Kim and Garlan [27], for example, apply the Alloy [28] analyzer to automatically verify architectural styles specified in ACME [29]. A similar approach comes from Wong et al. [30] which applies Alloy to the verification of architectural models. Zhang et al. [31] applied model checking techniques to verify architectural styles formulated in Wright#, an extension of Wright. Similarly, Marmsoler and Degenhardt [32] also apply model checking for the verification of design patterns. Another approach comes from Wirsing et al. [33] where the authors apply rewriting logic to specify and verify cloud-based architectures. While all these approaches focus on the verification of architectures and architectural patterns, they all apply automatic verification techniques. While this has many advantages, verification is limited to properties subject to automatic verification. Indeed, with our work we actually complement these approaches by providing an alternative approach based on, rather than automatic verification techniques.

Interactive Theorem Proving for Software Architectures. Another area of related work can be found in applications of to software architectures in general. Fensel and Schnogge [34], for example, apply the KIV interactive theorem prover to verify concrete architectures in the area of knowledge-based systems. Their work differs from our work in two main aspects. (i) While they focus on the verification of concrete architectures, we propose an approach to verify architectural patterns. (ii) While they focus on the verification of static architecture, our approach allows for the verification of dynamic architectures. Thus, we complement their work by providing a more general approach. More recently, some attempts were made to apply to the verification of architectural connectors. Li and Sun [35], for example, apply the Coq proof assistant to verify connectors specified in Reo [36]. With our work we complement their approach since we focus on the verification of patterns, rather than connectors.

To summarize, to the best of our knowledge, this is the first attempt applying to the verification of architectural design patterns.

6 Conclusion

With this paper we presented a novel approach for the specification and verification of architecture design patterns. Therefore, we provide a methodology and corresponding specification techniques for the specification of patterns in terms of configuration traces. Then, we describe an algorithm to map a given specification to a corresponding Isabelle/HOL theory and show soundness of the algorithm. Our approach can be used to formally specify patterns in a hierarchical way. Using the algorithm, the specification can then be mapped to a corresponding Isabelle/HOL theory where the pattern can be verified using a pre-existing calculus. This is demonstrated by specifying and verifying versions of three architecture patterns: the singleton, the publisher subscriber, and the blackboard. Thereby, patterns were specified hierarchical and verification results for lower level patterns were reused for the verification of higher level patterns.

The proposed approach addresses the challenges for pattern verification identified in the introduction as follows:

	C1 Axiomatic specifications	C2 Dynamic aspects	C3 Hierarchical specifications
Specification	Model-theoretic semantics	Model of dynamic architectures	Structured specifications
Verification	Axiomatic reasoning	A calculus to support verification	Import of verification results

In order to achieve our overall vision of interactive, hierarchical pattern verification [37], future work is needed in two directions: We are currently working on an implementation of the approach for the eclipse modeling framework [38] where a pattern can be specified and a corresponding Isabelle/HOL theory can be generated using the algorithm presented in the paper. In a second step, we want to lift the verification to the architecture level, hiding the complexity of an interactive theorem prover and interpreting its output at the architecture level.

Acknowledgments. We would like to thank Veronika Bauer, Maximilian Junker, and all the anonymous reviewers of FASE 2018 for their comments and helpful suggestions on earlier versions of this paper. Parts of the work on which we report in this paper was funded by the German Federal Ministry of Education and Research (BMBF) under grant no. 01Is16043A.

References

1. Taylor, R.N., Medvidovic, N., Dashofy, E.M.: *Software Architecture: Foundations, Theory, and Practice*. Wiley Publishing, Chichester (2009)
2. Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., Stal, M.: *Pattern-Oriented Software Architecture: A System of Patterns*. Wiley, West Sussex (1996)
3. Shaw, M., Garlan, D.: *Software Architecture: Perspectives on an Emerging Discipline*, vol. 1. Prentice Hall, Englewood Cliffs (1996)
4. Wiedijk, F. (ed.): *The Seventeen Provers of the World*. LNCS (LNAI), vol. 3600. Springer, Heidelberg (2006). <https://doi.org/10.1007/11542384>
5. Marmosler, D., Gleirscher, M.: On activation, connection, and behavior in dynamic architectures. *Sci. Ann. Comput. Sci.* **26**(2), 187–248 (2016)
6. Marmosler, D., Gleirscher, M.: Specifying properties of dynamic architectures using configuration traces. In: Sampaio, A., Wang, F. (eds.) *ICTAC 2016*. LNCS, vol. 9965, pp. 235–254. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-46750-4_14
7. Marmosler, D.: *Dynamic architectures*. *Archive of Formal Proofs*, pp. 1–65. Formal proof development, July 2017
8. Marmosler, D.: *Towards a calculus for dynamic architectures*. In: Hung, D., Kapur, D. (eds.) *ICTAC 2017*. LNCS, vol. 10580. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-67729-3_6

9. Nipkow, T., Wenzel, M., Paulson, L.C. (eds.): Isabelle/HOL: A Proof Assistant for Higher-Order Logic. LNCS, vol. 2283. Springer, Heidelberg (2002). <https://doi.org/10.1007/3-540-45949-9>
10. Gordon, M.J., Milner, A.J., Wadsworth, C.P.: Edinburgh LCF: A Mechanised Logic of Computation. LNCS, vol. 78. Springer, Heidelberg (1979). <https://doi.org/10.1007/3-540-09724-4>
11. Berghofer, S., Wenzel, M.: Inductive datatypes in HOL — lessons learned in formal logic engineering. In: Bertot, Y., Dowek, G., Théry, L., Hirschowitz, A., Paulin, C. (eds.) TPHOLs 1999. LNCS, vol. 1690, pp. 19–36. Springer, Heidelberg (1999). https://doi.org/10.1007/3-540-48256-3_3
12. Wenzel, M.: Type classes and overloading in higher-order logic. In: Gunter, E.L., Felty, A. (eds.) TPHOLs 1997. LNCS, vol. 1275, pp. 307–322. Springer, Heidelberg (1997). <https://doi.org/10.1007/BFb0028402>
13. Wenzel, M.: Isabelle/Isar - a generic framework for human-readable proof documents. In: From Insight to Proof - Festschrift in Honour of Andrzej Trybulec vol. 10, no. 23, pp. 277–298 (2007)
14. Ballarin, C.: Locales and locale expressions in Isabelle/Isar. In: Berardi, S., Coppo, M., Damiani, F. (eds.) TYPES 2003. LNCS, vol. 3085, pp. 34–50. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-24849-1_3
15. Broy, M.: A logical basis for component-oriented software and systems engineering. *Comput. J.* **53**(10), 1758–1782 (2010)
16. Broy, M.: A model of dynamic systems. In: Bensalem, S., Lakhneck, Y., Legay, A. (eds.) ETAPS 2014. LNCS, vol. 8415, pp. 39–53. Springer, Heidelberg (2014). https://doi.org/10.1007/978-3-642-54848-2_3
17. Marmsoler, D.: On the semantics of temporal specifications of component-behavior for dynamic architectures. In: Eleventh International Symposium on Theoretical Aspects of Software Engineering. Springer (2017)
18. Broy, M.: Algebraic specification of reactive systems. In: Wirsing, M., Nivat, M. (eds.) AMAST 1996. LNCS, vol. 1101, pp. 487–503. Springer, Heidelberg (1996). <https://doi.org/10.1007/BFb0014335>
19. Wirsing, M.: Algebraic specification. In: van Leeuwen, J. (ed.) Handbook of Theoretical Computer Science, pp. 675–788. MIT Press, Cambridge (1990)
20. Manna, Z., Pnueli, A.: The Temporal Logic of Reactive and Concurrent Systems. Springer, New York (1992). <https://doi.org/10.1007/978-1-4612-0931-7>
21. Wenzel, M., et al.: The Isabelle/Isar reference manual (2004)
22. Marmsoler, D.: Isabelle/HOL theories for the singleton, publisher subscriber, and blackboard pattern. <http://www.marmsoler.com/docs/FASE18>
23. Allen, R.J.: A formal approach to software architecture. Technical report, DTIC Document (1997)
24. Attie, P., Baranov, E., Bliudze, S., Jaber, M., Sifakis, J.: A general framework for architecture composability. *Form. Asp. Comput.* **28**(2), 207–231 (2016)
25. Mavridou, A., Baranov, E., Bliudze, S., Sifakis, J.: Architecture diagrams: a graphical language for architecture style specification. In: Bartoletti, M., Henrio, L., Knight, S., Vieira, H.T. (eds.) Proceedings of the 9th Interaction and Concurrency Experience. ICE 2016, Heraklion, 8–9 June 2016. EPTCS, vol. 223, pp. 83–97 (2016)
26. Mavridou, A., Baranov, E., Bliudze, S., Sifakis, J.: Configuration logics: modelling architecture styles. In: Braga, C., Ölveczky, P.C. (eds.) FACS 2015. LNCS, vol. 9539, pp. 256–274. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-28934-2_14

27. Kim, J.S., Garlan, D.: Analyzing architectural styles with alloy. In: Proceedings of the ISSTA 2006 Workshop on Role of Software Architecture for Testing and Analysis, pp. 70–80. ACM (2006)
28. Jackson, D.: Alloy: a lightweight object modelling notation. *ACM Trans. Softw. Eng. Methodol. (TOSEM)* **11**(2), 256–290 (2002)
29. Garlan, D.: Formal modeling and analysis of software architecture: components, connectors, and events. In: Bernardo, M., Inverardi, P. (eds.) *SFM 2003*. LNCS, vol. 2804, pp. 1–24. Springer, Heidelberg (2003). https://doi.org/10.1007/978-3-540-39800-4_1
30. Wong, S., Sun, J., Warren, I., Sun, J.: A scalable approach to multi-style architectural modeling and verification. In: *Engineering of Complex Computer Systems*, pp. 25–34. IEEE (2008)
31. Zhang, J., Liu, Y., Sun, J., Dong, J.S., Sun, J.: Model checking software architecture design. In: *High-Assurance Systems Engineering*, pp. 193–200. IEEE (2012)
32. Marmosler, D., Degenhardt, S.: Verifying patterns of dynamic architectures using model checking. In: *Proceedings of the International Workshop on Formal Engineering approaches to Software Components and Architectures, FESCA@ETAPS 2017*, Uppsala, Sweden, 22 April 2017, pp. 16–30 (2017)
33. Wirsing, M., Eckhardt, J., Mühlbauer, T., Meseguer, J.: Design and analysis of cloud-based architectures with KLAIM and Maude. In: Durán, F. (ed.) *WRLA 2012*. LNCS, vol. 7571, pp. 54–82. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-34005-5_4
34. Fensel, D., Schnogge, A.: Using KIV to specify and verify architectures of knowledge-based systems. In: *Automated Software Engineering*, pp. 71–80, November 1997
35. Li, Y., Sun, M.: Modeling and analysis of component connectors in Coq. In: Fiadeiro, J.L., Liu, Z., Xue, J. (eds.) *FACS 2013*. LNCS, vol. 8348, pp. 273–290. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-07602-7_17
36. Arbab, F.: Reo: a channel-based coordination model for component composition. *Math. Struct. Comput. Sci.* **14**(03), 329–366 (2004)
37. Marmosler, D.: Towards a theory of architectural styles. In: *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering - FSE 2014*, pp. 823–825. ACM Press (2014)
38. Steinberg, D., Budinsky, F., Merks, E., Paternostro, M.: *EMF: Eclipse Modeling Framework*. Pearson Education, London (2008)

Open Access This chapter is licensed under the terms of the Creative Commons Attribution 4.0 International License (<http://creativecommons.org/licenses/by/4.0/>), which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.

