A Framework for Interactive Verification of Architectural Design Patterns in Isabelle/HOL

Diego Marmsoler✉ https://orcid.org/0000-0003-2859-7673

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

Abstract. Architectural design patterns capture architectural design experience and are an important tool in software engineering to support the conceptualization and analysis of architectures. They constrain different aspects of an architecture and usually guarantee some corresponding properties for architectures implementing them. Verifying such patterns requires proving that the constraints imposed by the pattern indeed lead to architectures which satisfy the corresponding guarantee. Due to the abstract nature of patterns, verification is often done by means of interactive theorem proving and requires detailed knowledge about the underlying model, limiting its application to experts of this model. Moreover, proving properties for different patterns usually involves repetitive proof steps, leading to proofs which are difficult to maintain. To address these problems, we developed a framework that supports the interactive verification of architectural design patterns in Isabelle/HOL. The framework implements a model for dynamic architectures as well as a corresponding calculus in terms of two Isabelle/HOL theories and consists of roughly 3 500 lines of Isabelle/HOL proof script. To evaluate our framework, we applied it for the verification of four different architectural design patterns and compared the overall amount of proof code to the code contributed by the framework. Our results suggest that the framework has the potential to significantly reduce the amount of proof code required for the verification of patterns and thus to address the problems mentioned above.

Keywords: Architectural Design Pattern, Interactive Theorem Proving, Architecture Verification, Configuration Trace, Co-inductive List, Isabelle/HOL.

1 Introduction

Architectural design patterns (ADPs) are an important tool in software engineering for the conceptualization and analysis of software systems. They capture architectural design experience and are regarded as the 'Grand Tool' for designing a software systems architecture [1]. ADPs usually constrain the design of an architecture: the types of components, the activation/deactivation of components of a certain type, and connections between active components. In return, they guarantee certain safety/liveness properties for architectures implementing the pattern [2]. Verifying ADPs requires to verify whether the constraints imposed by them indeed lead to the claimed guarantees. Due to the abstract nature of patterns, verification is often done by means of interactive theorem proving [3]. The corresponding process is summarized in Fig. 1: The specification of an ADP is given in terms of temporal logic formulæ to express the constraints on the behavior of components as well as on their activation/deactivation and interconnection. In addition, the pattern's guarantee is specified in terms of a temporal logic formula over

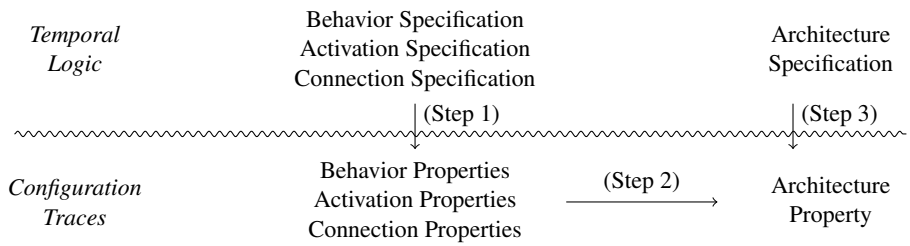


Fig. 1: Interactive verification of architectural design patterns.

the architecture as a whole. To verify the pattern, one has to interpret the specification at the model level in terms of different sets of configuration traces (Step 1) and show that the intersection of these properties leads to a property (Step 2) which corresponds to the claimed guarantee (Step 3).

Problem description. First attempts to verify different patterns revealed that reasoning about a pattern's specification imposes the following challenges:

- The interpretation of the specification (Step 1) and verification results (Step 3) requires deep knowledge about the model of configuration traces.
- The proof of the guarantee itself (Step 2) requires many repetitive steps which are similar for every ADP.

These problems have two negative consequences on the verification of ADPs:

- The required expertise limits the practical applicability of the approach: verification is restricted to experts of the model.
- The repetitive nature of the proofs increases the effort to verify a pattern in the first place as well as to maintain verification results in the long run.

Approach. In order to address the problems identified above, we developed a framework to support the interactive verification of patterns at a more abstract level: we implemented the model of configuration traces [4,5] as well as a corresponding calculus [6,7] to support reasoning about component behavior, in Isabelle/HOL [8]. The implementation consists of two theories amounting to roughly 3 500 lines of Isabelle/HOL proof script and is available online via the archive of formal proofs [9].

Evaluation. To evaluate the framework, we first applied it for the verification of four different ADPs: a variant of the *Singleton*, the *Publisher-Subscriber*, and the *Blackboard* pattern as well as a pattern for *Blockchain* architectures. Then, we calculated the overall number of proof lines for each of the patterns and compared it to the number of lines contributed by the framework. The corresponding theories are again available online as a separate entry in the archive of formal proofs [10] and consist of another 3 500 lines of Isabelle/HOL proof script.

Contributions. With this paper, we report on the results obtained from our development. Thus, its major contributions can be summarized as follows:

- It *describes* the *framework* itself: the major definitions and corresponding theorems as well as its interface in terms of an Isabelle/HOL locale [11].
- It *demonstrates* the *use of the framework* in terms of a running example.
- It *presents* and *discusses* the data obtained from the *evaluation* of the framework.

Overview. The remainder of the paper is organized as follows: In Sect. 2, we provide some background for our work. Therefore, we first describe our formal model of archi-

tectures (Sect. 2.1). Then, we introduce the Blackboard pattern as our running example (Sect. 2.2). Finally, we provide some general background on Isabelle/HOL for readers who are not familiar with it (Sect. 2.3). In Sect. 3, we then present our framework. Therefore, we first provide an overview of the major definitions and theorems and then we demonstrate its usage in terms of our running example. We continue with a discussion about the evaluation of the framework and the obtained results in Sect. 4. Finally, we discuss related work in Sect. 5 and conclude with a brief outlook and a description of next steps in Sect. 6.

2 Background

In the following, we provide some background for our work. Therefore, we first introduce the formal model of architectures on which our approach is based on. Then, we introduce a variant of the Blackboard pattern which serves as a running example. Finally, we provide some background on Isabelle/HOL for readers not familiar with it.

2.1 A Model of Dynamic Architectures

Since some architectural patterns involve dynamic aspects, such as component activation and deactivation, as well as architecture reconfiguration, our framework is based on a model of dynamic architectures. In our model, such an architecture is represented by *a set* of so-called *configuration traces* [4,5], i.e., streams [12] of architecture snapshots. An *architecture snapshot* represents the state of an architecture at some point in time: active components with their ports valuated by messages and connections between the ports of these components.

Figure 2 depicts the first three snapshots of a conceptual representation of an exemplary configuration trace: components are represented by gray rectangles and their ports by empty (input) and filled (output) circles, port valuations are represented by assignments of sets of messages (numbers or characters) to the corresponding port, and connections between ports are represented by solid lines between the ports.

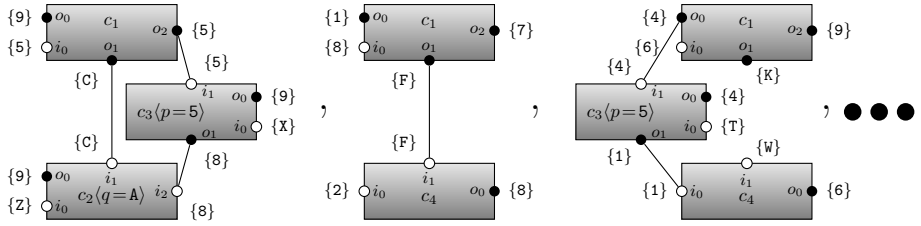


Fig. 2: Conceptual representation of a configuration trace.

Composition is modeled in terms of *behavior projection* which can be used to extract the behavior of a single component out of a given configuration trace. Given *valid* behavior for each component in terms of streams of valuations of the component's ports, composition of components results in a set of configuration traces, such that the behavior of each component (obtained through projection) is a valid behavior for that component.

2.2 A Pattern for Blackboard Architectures

Throughout the paper, we shall use a variant of the Blackboard pattern [1,2,13] to demonstrate our ideas. Blackboard architectures are usually found in systems which

work on some collaborative problem solving task. Thereby, it is desired to design an architecture which can solve a complex problem (such as solving a logical equation composed of several sub formulas connected by logical operators) by breaking it down into simpler sub problems (such as solving the corresponding sub formulas) which can be solved and assembled to a solution for the original problem.

In the following, we specify the pattern in terms of sets of configuration traces. Therefore, we first specify the types of involved messages in terms of abstract data types [14,15]. Then, we specify component types in two steps: first, we graphically specify a set of interfaces using so-called *configuration diagrams* [16]. Then, we specify assertions about the behavior of components in terms of so-called *behavior trace assertions*, i.e., linear temporal logic formulæ [17] using port names as free variables. Finally, we add architectural assertions to specify component activation and deactivation as well as connection reconfiguration. Architectural assertions are specified using so-called *configuration trace assertions*, i.e., linear temporal logic formulæ using component variables and some designated predicates to express component activation ($\|\_\|$) and connections between ports of components ($\_.\_ \rightsquigarrow \_.\_$).

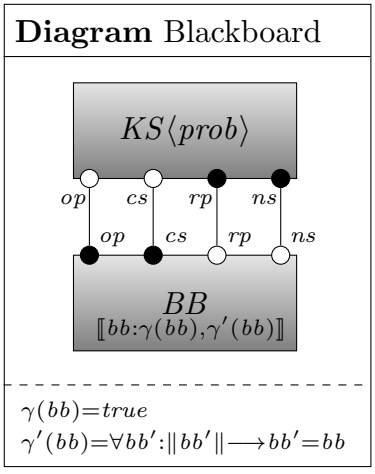
Data types. Blackboard architectures work with *problems* and *solutions* for them. Figure 3a provides an algebraic specification for the corresponding data types. We denote by PROB the set of all problems and by SOL the set of all solutions. Complex problems consist of *subproblems* which can be complex themselves. To solve a problem, its subproblems have to be solved first. Therefore, we assume the existence of a *subproblem relation* $\prec \subseteq$ PROB $\times$ PROB. For complex problems, this relation may not be known in advance. Indeed, one of the benefits of a Blackboard architecture is that a problem can be solved also without knowing this relation in advance. However, the subproblem relation has to be well-founded (a partial order with no infinite decreasing chains) (Eq. (1)) 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 for a problem

DTSpec ProbSol	**imports** SET
$\prec$:	PROB $\times$ PROB
solve:	PROB $\rightarrow$ SOL
well-founded($\prec$)	(1)

(a) Data Type Specification.

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

(b) Port Specification.



(c) Configuration Diagram.

Fig. 3: Specification of a Blackboard architecture.

is always unique. Thus, we postulate the existence of a function $solve\colon \texttt{PROB} \to \texttt{SOL}$ which assigns the *correct* solution to each problem. Note, however, that this function is not known in advance and it is one of the reasons of using this pattern to calculate this function.

Component types. Two types of components are common for Blackboard architectures: blackboards and knowledge sources. The corresponding interfaces are specified by the configuration diagram depicted in Fig. 3c. The types of data which can be exchanged through each of the ports is given by the corresponding port specification (Fig. 3b). Since each component of type knowledge sources can solve only certain problems, knowledge sources are parametrized by a problem $prob$. In the following, we specify the behavior of components of each of the two types.

Blackboard components. A Blackboard provides the *current state* towards solving the original problem and forwards problems and solutions from knowledge sources. Fig. 4 provides a specification of the blackboard's behavior in terms of three behavior trace assertions[1]:

- if a solution to a subproblem is received on its input, then it is eventually provided at its output (Eq. 2).
- if solving a problem requires a set of subproblems to be solved first, those problems are eventually provided at its output (Eq. (3)).
- a problem is provided as long as it is not solved (Eq. (4)).

BSpec Blackboard		**for** BB **of** Blackboard
flex	p:	PROB
	P:	PROB SET
rig	p':	PROB
	s':	SOL
$\Box\Big((p', s') \in ns \longrightarrow \Diamond\big((p', s') \in cs\big)\Big)$		(2)
$\Box\Big((p, P) \in rp \longrightarrow \big(\forall p' \in P\colon\ (\Diamond(p' \in op))\big)\Big)$		(3)
$\Box\Big(p' \in op \longrightarrow \big(p' \in op\ \mathcal{W}\ (p', solve(p')) \in cs\big)\Big)$		(4)

Fig. 4: Specification of behavior for blackboard components.

Knowledge source components. A knowledge source receives open problems via op and solutions for other problems via cs. It might contribute to the solution of the original problem by solving subproblems. Fig. 5 provides a specification of the knowledge source's behavior in terms of three behavior trace assertions:

- if a knowledge source gets correct solutions for all the required subproblems, then it solves the problem eventually (Eq. (5)).
- to solve a problem, a knowledge source requires solutions only for smaller problems (Eq. (6)).

[1] Note that the specification uses *flexible* and *rigid* variables: while the former are newly interpreted at each point in time, the latter keep their value over time. Moreover, it uses the *weak* until operator which is defined as follows: $\gamma'\ \mathcal{W}\ \gamma \quad \stackrel{\text{def}}{=} \quad \Box(\gamma') \vee (\gamma'\ \mathcal{U}\ \gamma)$.

– a knowledge source will eventually communicate if it is able to solve a problem (Eq. (7)).

BSpec Knowledge Source		**for** $KS\langle prob\rangle$ **of** Blackboard
flex	p:	PROB
	P:	$\wp$(PROB)
rig	p':	PROB
$\Box\Big(\forall(prob, P) \in rp\colon \big((\forall p' \in P\colon \Diamond(p', solve(p')) \in cs)$		
$\longrightarrow \Diamond(prob, solve(prob)) \in ns\big)\Big)$		(5)
$\Box\Big(\forall(prob, P) \in rp\colon \forall p \in P\colon p \prec prob\Big)$		(6)
$\Box\Big(prob \in op \longrightarrow \Diamond(\exists P\colon (prob, P) \in rp)\Big)$		(7)

Fig. 5: Specification of behavior for knowledge source components.

Architectural constraints. Architectural constraints for the Blackboard pattern are described by the configuration diagram in Fig. 3c: The '$[\![bb\colon \gamma(bb), \gamma'(bb)]\!]$' annotation for a blackboard interface provides lower and upper bounds for the activation of blackboard components

– With $\gamma(bb) = true$, we require that a blackboard component is always activated.
– With $\gamma'(bb) = \forall bb'\colon \|bb'\| \longrightarrow bb' = bb$, we require that a blackboard component is unique.

Thus, we require a *unique* blackboard component to be always activated. The absence of annotations for KS interfaces, on the other hand, allows knowledge source components to be de-/activated over time. The *solid* connections between the ports denote a constraint requiring that the ports of a knowledge source component are connected with the corresponding ports of a blackboard component as depicted, whenever both components are active. Note that many knowledge sources may be active at each point in time, in which case every knowledge source is connected to the blackboard as depicted in the diagram.

The constraints introduced by the configuration diagram are refined by an additional configuration trace assertions provided in Fig. 6: By Eq. (8) we require that whenever a knowledge source obtains a request to solve a problem which it is able to solve, the knowledge source will stay active until that problem is solved ($ks'_{p'}$ denotes a knowledge source which is able to solve problem p').

ASpec Blackboard		**for** Blackboard
rig	ks':	$KS\langle prob\rangle$
	p':	PROB
$\Box\Big(\|ks'_{p'}\| \wedge p' \in ks'.op \longrightarrow \big(\|ks'_{p'}\| \,\mathcal{W}\, \|ks'_{p'}\| \wedge (p', solve(p')) \in ks'.ns\big)\Big)$		(8)

Fig. 6: Specification of activation constraints for Blackboard architectures.

2.3 Isabelle/HOL

Isabelle is an LCF-style [18] theorem prover based on Standard ML [19]. 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. To support the specification of theories, Isabelle/HOL provides tools for the specification of (co)datatypes [20] and (co)recursive functions. To support the verification of theorems, Isabelle/HOL provides a structured proof language called Isabelle/Isar [21] and a set of logical reasoners to verify correctness of single proof steps. For the development of our framework, two additional features of Isabelle are important: coinductive lists and locales.

Coinductive lists. In order to implement the model of configuration traces in Isabelle/HOL, we relied on Lochbihler's theory of coinductive (lazy) lists [22]. In his theory, Lochbihler formalized lazy lists using Isabelle/HOL's coinductive datatype package and provides definitions and properties for many important concepts such as filtering elements from infinite lists to create new lists.

Locales. In Isabelle, modularization of theories is supported through the notion of locales [11]. A locale consists of a list of type and function parameters and corresponding assumptions about them. Locales can extend other locales and may be instantiated by concrete definitions of the corresponding parameters.

3 IPV: A Framework for Interactive Pattern Verification

Figure 7 provides an overview of our framework for the interactive verification of architectural design patterns: The framework consists of two Isabelle/HOL theories which are available through the archive of formal proofs [9]:

`Configuration_Traces` imports theory `Coinductive_List` and provides a formalization of the model described in Sect. 2.1 in terms of lazy lists.

`Dynamic_Architecture_Calculus` imports theory `Configuration_Traces`, provides operators for the specification of component behavior, and implements a calculus to reason about component behavior [7] specified using these operators.

Moreover, the framework provides an interface to these theories in terms of an Isabelle/HOL locale [11] `dynamic_component`. The locale requires definitions for certain concepts of the model (Step 1) and then provides customized operators for the specification of component behavior (Step 2) and rules to support reasoning about such specifications (Step 3). A pattern theory may use the framework by instantiating the locale for every type of component involved in the pattern. Then, the behavior of each component type is specified using the specification operators provided by the corresponding instantiation. Moreover, activation and connection constraints are specified for components of the different types. Finally, the pattern can be verified by using the verification rules provided by the framework.

In the following, we demonstrate the different steps in more detail in terms of our running example. Thereby, we specify the Blackboard pattern introduced in Sect. 2.2 in Isabelle/HOL and verify a characteristic property of such architectures using the verification rules provided by the framework. The corresponding Isabelle/HOL script is provided in this paper's electronic supplementary material [23].

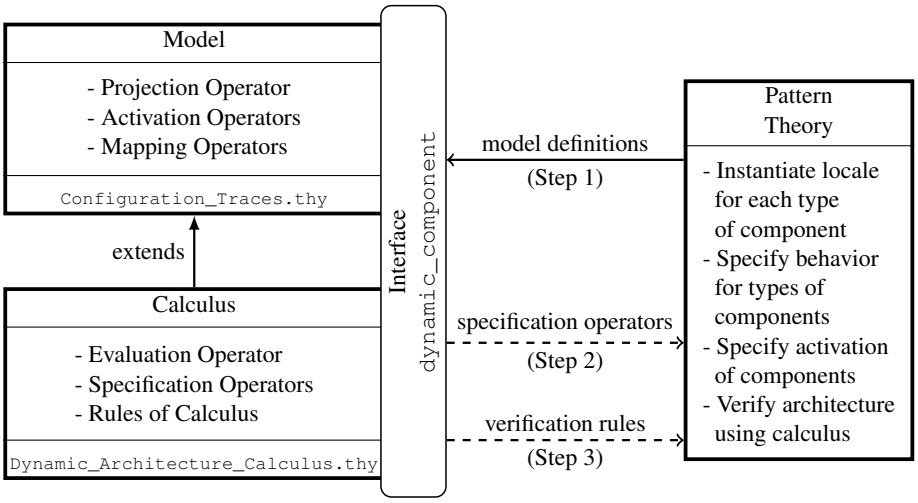


Fig. 7: Overview of IPV Framework.

3.1 Creating the Theory

As a first step, we create a new Isabelle/HOL theory which imports theory `Dynamic_Architecture_Calculus` [9] from the Archive of Formal Proofs:

theory *Blackboard* **imports** *DynamicArchitectures.Dynamic-Architecture-Calculus*

Note that importing theory `Dynamic_Architecture_Calculus` is crucial here, since it provides access to locale `dynamic_component` which is used in Sect. 3.3 to specify the pattern.

3.2 Specifying Data Types

Next, we specify Isabelle/HOL datatypes for the data types required by the Blackboard pattern (specified in Fig. 3a):

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

First, we introduce a new type $PROB$ for the problems to be solved by the architecture. Moreover, we specify a corresponding constant sb which relates such problems with corresponding subproblems. Then, we require the well-foundedness constraint for subproblem relations (specified by Eq. (1) of Fig. 3a) by a corresponding axiom $sbWF$. Finally, we introduce another type SOL for solutions to the problems and a corresponding constant $solve$ which assigns the correct solution to a given problem.

3.3 Specifying Component Types

Now, we specify the types of components involved in a Blackboard architecture. Therefore, we create an Isabelle/HOL locale for the pattern which imports locale `dynamic_component` (from the framework) for each type of component. Then,

we add locale parameters for each type of port according to the pattern's port specification. Finally, we use the operators provided by the framework to specify the assertions about the behavior of components of the corresponding types.

Creating the pattern's locale. According to Sect. 2.2, a Blackboard architecture has two types of components: blackboards and knowledge sources. Thus, we create a locale `blackboard` with two instantiations of locale `dynamic_component`: *bb* and *ks*, respectively.

locale *blackboard* =
bb: *dynamic-component bbcmp bbactive* +
ks: *dynamic-component kscmp ksactive*
for *bbactive* :: *'bid* ⇒ *cnf* ⇒ *bool* (‖-‖-)
and *bbcmp* :: *'bid* ⇒ *cnf* ⇒ *'BB* (*σ*-(-))
and *ksactive* :: *'kid* ⇒ *cnf* ⇒ *bool* (‖-‖-)
and *kscmp* :: *'kid* ⇒ *cnf* ⇒ *'KS* (*σ*-(-)) +

Note that locale `dynamic_component` requires two parameters for each type of component: a function to denote activation of a component of the corresponding type within a given architecture snapshot, and another function to obtain a certain component from a given architecture snapshot. Thus, we specify corresponding functions to denote activation and component selection for blackboards ($bbactive$ and $bbcmp$) as well as for knowledge sources ($ksactive$ and $kscmp$) and pass them to the corresponding locale import. To foster readability, we also provide concrete syntax for these functions: $\|\_\|$ for activation and $\sigma(\_)$ for selection.

Importing the locale for blackboards and knowledge sources provides us with custom specification mechanisms for these types of components:

- Evaluation functions $bb.eval$ and $ks.eval$, to evaluate component behavior specifications for blackboards and knowledge sources, respectively.
- Specification operators for common linear temporal logic operators such as next (*bb.nxt*/*ks.nxt*), globally (*bb.glob*/*ks.glob*), eventually (*bb.evt*/*ks.evt*), until (*bb.until*/*ks.until*), and weak until (*bb.wuntil*/*ks.wuntil*).

Moreover, importing the locale also provides us with rules to support reasoning about specifications involving these operators (in terms of Isabelle/HOL lemmata). In essence, the framework provides introduction and elimination rules for all the temporal operators combined with corresponding specifications of component activation (this amounts to roughly 35 rules for each type of component).

Specifying ports and parameters. As described in Sect. 2.2, each component type specifies 4 ports to exchange data with its environment. Thus, for each instance of the port types specified in Fig. 3b by one of the component types, we create one corresponding locale parameter:

fixes *bbrp* :: *'BB* ⇒ (*PROB* × *PROB set*) *set*
and *ksrp* :: *'KS* ⇒ *PROB* × *PROB set*
and *bbns* :: *'BB* ⇒ (*PROB* × *SOL*) *set*
...

The parameters are modeled as functions which take a snapshot of a component of the corresponding type ('BB or 'KS) and return a set of elements according to the ports data type.

As mentioned in Sect. 2.2, each knowledge source is parametrized by a problem which it can solve. Thus, we first add a corresponding locale parameter which associates a problem with each knowledge source:

and *prob* :: *'kid* ⇒ *PROB*

Moreover, we add a locale assertion which ensures that at least one knowledge source exists for each possible problem:

assumes
ks1: ∀ *p*. ∃ *ks*. *p*=*prob ks*

Specifying component behavior. Next we can specify the assertions about the behavior of the blackboard or knowledge sources as specified in Fig. 4 and Fig. 5, respectively. Note that assumptions about component behavior are specified in terms of behavior trace assertions (without considering component activation or deactivation). Thus, the specification of the corresponding locale assumptions need to be formulated using the temporal logic operators provided by the framework and described above:

and *bhvbb1*: ⋀*t t' bid p s*. ⟦*t* ∈ *arch*⟧ ⟹ *bb.eval bid t t' 0*
(□$_b$ (*bb.ba* (λ *bb*. (*p*,*s*)∈*bbns bb*) ⟶^{b} (◇$_b$ (*bb.ba* (λ *bb*. (*p*,*s*) ∈ *bbcs bb*)))))
...
and *bhvks1*: ⋀*t t' kId p P*. ⟦*t*∈*arch*; *p* = *prob kId*⟧ ⟹ *ks.eval kId t t' 0*
(□$_b$ ((*ks.ba* (λ *ks*. (*p*, *P*) = *ksrp ks*)) ∧^{b}
(∀$_b$ *q*. ((*ks.pred* (*q*∈*P*)) ⟶^{b} (◇$_b$ (*ks.ba* (λ *ks*. (*q*,*solve*(*q*)) ∈ *kscs ks*)))))
⟶^{b} (◇$_b$ (*ks.ba* (λ *ks*. (*p*,*solve*(*p*)) ∈ *ksns*(*ks*))))))
...

bhvbb1, for example, formalizes Eq. (2): with $bb.eval\ bid\ t\ t'\ 0$, we require that the subsequent formula is to be interpreted for a component of type blackboard (since we are using the blackboard variant of *eval*) with a certain identifier bid for a configuration trace t at time point 0. The specification of the formula itself then uses the corresponding temporal operators from the framework: $bb.glob$ for globally, $bb.evt$ for eventually, and $bb.ass$ for basic assertions about the state of a blackboard component. Note that such basic assertions specify states of concrete components in terms of a function which take a component's state as input and returns whether it is valid or not for the time point given by the surrounding temporal specification.

bhvks1 formalizes Eq. (5): again, we use the evaluation function to access the framework and use the corresponding temporal operators to formulate the property. Since this time, however, we are formalizing a specification for knowledge sources, we need to use the corresponding instantiation for knowledge sources (ks).

3.4 Specifying Architectural Constraints

Finally, we can finalize the specification of the pattern by adding architectural constraints imposed by the pattern as described in Sect. 2.2. Since architectural assertions can be directly expressed over configuration traces, we do not need to use the operators provided by the framework for their specification.

Specifying assertions about component activation. First, we specify assertions about the activation of blackboards and knowledge sources:

and *alwaysActive*: $\bigwedge k.\ \exists id::'bid.\ \|id\|_k$
and *unique*: $\exists id.\ \forall k.\ \forall id'::'bid.\ \|id'\|_k \longrightarrow id = id'$
and *actks*: $\bigwedge t\ n\ kid\ p.\ [\![t \in arch;\ \|kid\|_{t\ n};\ p{=}prob\ kid;\ p{\in}ksop\ (\sigma_{kid}(t\ n))]\!]$
$\Longrightarrow (\exists n'{\geq}n.\ \|kid\|_{t\ n'} \wedge (p, solve\ p) \in ksns\ (\sigma_{kid}(t\ n')) \wedge (\forall n''{\geq}n.\ n''{<}n' \longrightarrow \|kid\|_{t\ n''}))$
$\vee\ (\forall n'{\geq}n.\ (\|kid\|_{t\ n'} \wedge (\neg(p, solve\ p) \in ksns\ (\sigma_{kid}(t\ n')))))$

The first two assumptions formalize the activation of blackboard components as specified by the configuration diagram in Fig. 3c: with $alwaysActive$ and $unique$ we require a *unique* blackboard component to be *always* activated. The third assertion specifies activation of knowledge source components as expressed by the configuration trace assertion shown in Fig. 6: With $actks$ we require that, whenever a knowledge source obtains a request to solve a problem which it is able to solve, the knowledge source will stay active until that problem is solved.

Specifying assertions about component connection. Finally, we specify assertions about connections between component ports:

and *conn1*: $\bigwedge k\ bid.\ \|bid\|_k \Longrightarrow bbns\ (\sigma_{bid}(k)) = (\bigcup kid{\in}\{kid.\ \|kid\|_k\}.\ ksns\ (\sigma_{kid}(k)))$
...

Connections between component ports are modeled in terms of conditional equalities between the connected ports (the condition requires the components to be activated). $conn1$, for example, specifies a required connection between ports ns of a blackboard and ns of a knowledge source.

3.5 Verifying Blackboard Architectures

Finally, we can use the framework to verify the specification of the Blackboard pattern. Therefore, we first specify a desired property for Blackboard architectures as an Isabelle/HOL theorem and prove it using the verification rules provided by the framework.

A characteristic property for Blackboard architectures. In the following, we specify a characteristic property for Blackboard architectures as an Isabelle/HOL theorem:

theorem *pSolved*:
fixes t **and** t'::*nat* $\Rightarrow$ '*BB* **and** p **and** t''::*nat* $\Rightarrow$ '*KS*
assumes $t{\in}arch$ **and** $\forall n.\ \forall p{\in}bbop(\sigma_{the\text{-}bb}(t\ n)).\ \exists n'{\geq}n.\ \|sKs\ p\|_{t\ n'}$
shows $\forall n.\ p{\in}bbop(\sigma_{the\text{-}bb}(t\ n)) \longrightarrow (\exists m{\geq}n.\ (p,solve(p)) \in bbcs\ (\sigma_{the\text{-}bb}(t\ m)))$

The property states that, if for each open (sub-)problem, a knowledge source which is able to solve the corresponding problem will be eventually activated, then the architecture guarantees that the original problem is indeed solved.

Verifying the property. In the following, we demonstrate how the rules of the framework can be used to support the verification process. Therefore, we present a small excerpt from the overall proof of the above property[2]. However, since the proof relies on rule evtEA, we first briefly describe this rule and then we show how it was used in the proof.

[2] The full proof is provided in [23].

Eventually elimination. In general, the rule has the following form:

evtEA

$$\frac{(t, t', n) \overset{t}{k}\models_{(c)} \Diamond\gamma}{\exists n' \geq \langle c \overset{n}{\vee} t\rangle \colon (t, t', n') \overset{t}{k}\models_{(c)} \gamma} \; \exists i \geq n \colon \|c\|_{t(i)}$$

It allows to eliminate an eventually operator from a behavior specification $\Diamond\gamma$ for a component c at time point n and conclude that γ holds sometimes after the last activation (before n) of component c. However, in order to be applied, the rule requires that component c is again activated in the future.

Proof excerpt. In the following excerpt we show how the above rule is used to eliminate the eventually operator available in locale assumption $bhvks1$, obtained from Eq. (5):

```
...
ultimately have ks.eval (sKs p) t t'' n_r (◇_b (ks.ba (λ ks. (p,solve(p))∈ksns(ks))))   1
 using ks.impE[of sKs p t t'' n_r] by blast                                              2
with ‹∃ i≥n_r. ‖sKs p‖_t i› obtain n_s where n_s≥⟨sKs p → t⟩n_r and                       3
 (∃ i≥n_s. ‖sKs p‖_t i ∧ (∀ n''≥⟨sKs p ⇐ t⟩n_s. n'' ≤ ⟨sKs p → t⟩n_s ⟶                    4
  ks.eval (sKs p) t t'' n'' (ks.ba (λ ks. (p,solve(p))∈ksns(ks))))) ∨                    5
 ¬ (∃ i≥n_s. ‖sKs p‖_t i) ∧ ks.eval (sKs p) t t'' n_s (ks.ba (λ ks. (p,solve(p))∈ksns(ks)))  6
 using ks.evtEA[of n_r sKs p t] by blast                                                 7
...
```

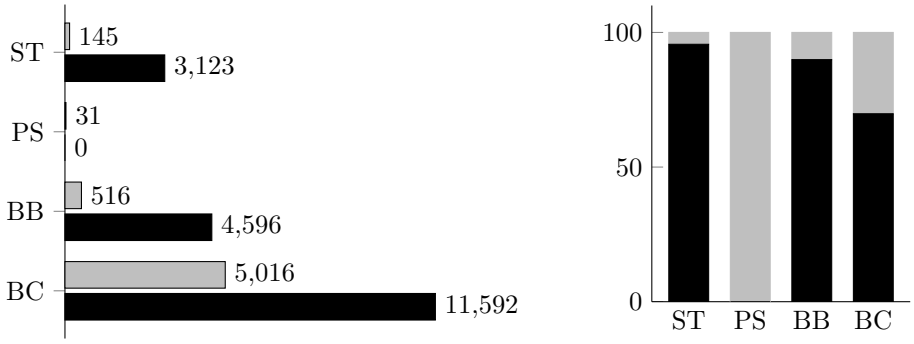
The excerpt starts with the fact that for a knowledge source with identifier $sKs\ p$, at some time point n_r, problem p and its solution $solve(p)$ are eventually provided at $sKs\ p's$ output port ns (lines 1 and 2). It then uses rule evtEA to obtain time point n_s for which problem p and its solution $solve(p)$ are actually provided at $sKs\ p's$ output port ns (lines 3 - 7). In order to do so, it uses another fact, which ensures that component $sKs\ p$ is indeed eventually activated, to discharge the side condition of rule evtEA (line 3).

4 Evaluation

In order to evaluate the framework, we used it to specify and verify four architectural design patterns: versions of the Singleton, Publisher-Subscriber, and Blackboard pattern, as well as a pattern for Blockchain architectures. The corresponding Isabelle/HOL theories are available via the archive of formal proofs and can be accessed online [10].

To investigate the frameworks potential to support the verification of architectural design patterns, we then calculated the *normalized* amount of proof code for the verification of each pattern and classified it into proof code attributed by the framework and additional proof code specific to the pattern. The collected raw data is available online [23] and summarized in Fig. 8. Fig. 8a depicts the absolute amount of framework code (black) and proof code specific to the pattern (gray) for each of the four patterns. It already suggests that the framework contributes a significant amount of proof code to the overall verification. To investigate this suspicion in more detail, Fig. 8b depicts the relative amount of framework code vs. proof code specific to the pattern. It shows that, except for the Publisher-Subscriber pattern, the amount of proof code contributed by the framework amounts to at least two-thirds of the overall proof code used to verify the patterns. The data obtained for the Publisher-Subscriber pattern can be explained by the absence of behavioral constraints imposed by the pattern. For the pattern was specified

in a way such that it requires only certain restrictions about the activation of components and connection between them, but no restrictions on the components actual behavioral was imposed. In summary, the data suggests that for patterns which do indeed involve constraints about component behavior, the proposed framework has the potential for significant reductions of the proof code required to verify them.



(a) Comparison of absolute values. (b) Relative comparision.

Fig. 8: Contribution of framework to overall verification.

5 Related Work

To the best of our knowledge, the framework presented in this paper is the first of its kind. Nevertheless, related work can be found in two related areas: applications of interactive theorem proving to software architectures and formalizations of temporal logics.

5.1 Interactive Theorem Proving for Software Architectures

Over the last decades, some attempts were made to apply interactive theorem proving to software architectures. One of the first attempts in this direction was done by Bergner [24]. The author proposes an approach to specify component networks and verify whether a given (runtime) component network satisfies its specification. The approach was implemented in Spectrum [25], a functional programming language which allows for axiomatic specifications of functions. Another approach comes from Fensel and Schnogge [26], which apply the KIV interactive theorem prover [27] to verify concrete architectures in the area of knowledge-based systems. Another example in this areas is the work of Spichkova [28] which provides a mapping from a FOCUS [29] specification to a corresponding Isabelle/HOL [8] theory. More recently, some attempts were made to apply interactive theorem proving to the verification of architectural connectors. Li and Sun [30], for example, apply the Coq proof assistant [31] to verify connectors specified in Reo [32].

While all these approaches apply interactive theorem proving to the verification of different aspects of software architectures, there is one major difference to our work: The above approaches mainly focus on the specification of static architectures. However, as argued in the introduction of this paper, dynamic architectures are becoming increasingly important. Thus, the work presented in this paper, complements these approaches by extending their scope to dynamic architectures.

5.2 Formalization of Temporal Logic

The framework proposed in this paper uses temporal logics as means to specify the behavior of dynamic components. Thus, formalizations of temporal logics in Isabelle/HOL represent another source for related work. First attempts in this direction focused on the formalization of Lamport's Temporal Logic of Actions [33]. An initial formalization of TLA is provided by Merz [34]. Then, Grov and Merz [35] elaborated on that work and formalized TLA* [36] in Isabelle/HOL. Later on, a formalization of temporal interval logic for real time systems is described by Mattolini and Nese [37]. A first formalization of LTL [17] in Isabelle/HOL is provided by Schimpf et al. [38] and then refined by Sickert [39].

While the above approaches all provide valuable insights into the process of formalizing temporal logics, the scope of this work is different: we are interested in combining a given temporal logic specification with a specification of component activations. To this end, we provide a calculus in terms of a set of rules which allows to reason about temporal specifications taking into consideration that states may be active or not. Thus, with our work we also complement existing work in this area.

6 Conclusion

In this paper, we described our results obtained by implementing a framework for the interactive verification of architectural design patterns (ADPs) in Isabelle/HOL:

- We described the major definitions and corresponding theorems as well as the interface to the framework.
- We demonstrate usage of the framework in terms of a running example: a dynamic version of the Blackboard pattern.
- We present and discuss the framework's evaluation in which the framework was used to specify and verify four different ADPs: a variant of the *Singleton*, the *Publisher-Subscriber*, and the *Blackboard* pattern as well as a pattern for *Blockchain* architectures.

Results. Our evaluation showed that for those patterns which involve the specification of component behavior, the framework contributed *at least* 75% of the proof code required for their verification. Moreover, the evaluation also suggested that using the framework allows reasoning at a more abstract level, thus reducing the amount of knowledge of the underlying model required for the verification of ADPs.

Implication. Based on our results, we conclude that the framework proposed in this paper has the potential to significantly reduce the amount of proof code required to verify ADPs, thus reducing the effort to develop and maintain verification results for ADPs. Moreover, reducing the necessary knowledge of the underlying model may increase practical applicability of the interactive verification of ADPs.

Vision and Outlook. Our overall research aims towards bringing interactive theorem proving closer to the software architecture community [40]. With the work presented in this paper, we provide another, important cornerstone towards this overall research agenda. However, additional work remains to be done in order to fully achieve our overall goal. In particular, future work should focus on the development of tools to support the interactive verification of patterns. Therefore, we are currently working on an implementation of our approach in Eclipse/EMF which uses the framework presented in this paper to support the interactive verification of ADPs.

Acknowledgments. We would like to thank all the people from the Isabelle mailing-list for their fast support. In particular, we would like to thank Andreas Lochbihler for his valuable support. Moreover, we would like to thank Ondřej Kunčar, Veronika Bauer, and all the anonymous reviewers of ICFEM 2018 for their comments and helpful suggestions on earlier versions of this paper.

References

1. Taylor, R.N., Medvidovic, N., Dashofy, E.M.: Software Architecture: Foundations, Theory, and Practice. Wiley Publishing (2009)
2. Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., Stal, M.: Pattern-Oriented Software Architecture: A System of Patterns. Wiley West Sussex, England (1996)
3. Marmsoler, D.: Hierarchical specication and verication of architecture design patterns. In: Fundamental Approaches to Software Engineering, FASE 2018, Thessaloniki, Greece, April 14-20, 2018, Proceedings. (2018)
4. Marmsoler, D., Gleirscher, M.: Specifying properties of dynamic architectures using configuration traces. In: International Colloquium on Theoretical Aspects of Computing. Springer (2016) 235–254
5. Marmsoler, D., Gleirscher, M.: On activation, connection, and behavior in dynamic architectures. Scientific Annals of Computer Science **26**(2) (2016) 187–248
6. 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)
7. Marmsoler, D.: Towards a calculus for dynamic architectures. In Hung, D.V., Kapur, D., eds.: Theoretical Aspects of Computing, ICTAC 2017, Hanoi, Vietnam, October 23-27, 2017, Proceedings. Volume 10580 of Lecture Notes in Computer Science., Springer (2017) 79–99
8. Nipkow, T., Paulson, L.C., Wenzel, M.: Isabelle/HOL: a proof assistant for higher-order logic. Volume 2283. Springer Science & Business Media (2002)
9. Marmsoler, D.: Dynamic architectures. Archive of Formal Proofs (July 2017) `http://isa-afp.org/entries/DynamicArchitectures.html`.
10. Marmsoler, D.: A theory of architectural design patterns. Archive of Formal Proofs (March 2018) `http://isa-afp.org/entries/Architectural_Design_Patterns.html`.
11. Ballarin, C.: Locales and locale expressions in isabelle/isar. Lecture notes in computer science **3085** (2004) 34–50
12. Broy, M.: A logical basis for component-oriented software and systems engineering. The Computer Journal **53**(10) (February 2010) 1758–1782
13. Shaw, M., Garlan, D.: Software Architecture: Perspectives on an Emerging Discipline. Volume 1. Prentice Hall Englewood Cliffs (1996)
14. Broy, M.: Algebraic specification of reactive systems. In: Algebraic Methodology and Software Technology, Springer, Springer Berlin Heidelberg (1996) 487–503
15. Wirsing, M.: Algebraic specification. In van Leeuwen, J., ed.: Handbook of Theoretical Computer Science (Vol. B). MIT Press, Cambridge, MA, USA (1990) 675–788
16. Marmsoler, D., Degenhardt, S.: Verifying patterns of dynamic architectures using model checking. In: Formal Engineering approaches to Software Components and Architectures, FESCA@ETAPS 2017, Uppsala, Sweden, 22nd April 2017. (2017) 16–30
17. Manna, Z., Pnueli, A.: The Temporal Logic of Reactive and Concurrent Systems. Springer New York (1992)
18. Gordon, M., Milner, R., Wadsworth, C.: Edinburgh LCF: A Mechanized Logic of Computation. 1 edn. Volume 78 of Lecture Notes in Computer Science. Springer-Verlag Berlin Heidelberg (1979)

19. Milner, R., Tofte, M., Harper, R.: The definition of Standard ML. MIT Pr., Cambridge, Mass. [u.a.] (1990) Literaturverz. S. [87] - 89.
20. Blanchette, J.C., Hölzl, J., Lochbihler, A., Panny, L., Popescu, A., Traytel, D.: Truly modular (co) datatypes for isabelle/hol. In: International Conference on Interactive Theorem Proving, Springer (2014) 93–110
21. Wenzel, M.: Isabelle/isar – a generic framework for human-readable proof documents. From Insight to Proof – Festschrift in Honour of Andrzej Trybulec **10**(23) (2007) 277–298
22. Lochbihler, A.: Coinduction. The Archive of Formal Proofs. http://afp.sourceforge.net/entries/Coinductive.shtml (2010)
23. Marmsoler, D.: A Framework for Interactive Verification of Architectural Design Patterns in Isabelle/HOL. Electronic Supplementary Material. `http://www.marmsoler.com/docs/ICFEM18/`
24. Bergner, K.: Spezifikation großer Objektgeflechte mit Komponentendiagrammen. PhD thesis, Technische Universität München (1996)
25. Broy, M., Facchi, C., Grosu, R., et al.: The requirement and design specification language spectrum – an informal introduction. Technical report, Technische Universität München (1993)
26. Fensel, D., Schnogge, A.: Using kiv to specify and verify architectures of knowledge-based systems. In: Automated Software Engineering. (November 1997) 71–80
27. Reif, W.: The kiv-approach to software verification. KORSO: Methods, Languages, and Tools for the Construction of Correct Software (1995) 339–368
28. Spichkova, M.: Specification and seamless verification of embedded real-time systems: FOCUS on Isabelle. PhD thesis, Technical University Munich, Germany (2007)
29. Broy, M., Stolen, K.: Specification and development of interactive systems: focus on streams, interfaces, and refinement. Springer Science & Business Media (2012)
30. Li, Y., Sun, M.: Modeling and analysis of component connectors in coq. In Fiadeiro, J.L., Liu, Z., Xue, J., eds.: 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., Springer (2013) 273–290
31. Bertot, Y., Castéran, P.: Interactive theorem proving and program development: Coq'Art: the calculus of inductive constructions. Springer Science & Business Media (2013)
32. Arbab, F.: Reo: a channel-based coordination model for component composition. Mathematical structures in computer science **14**(03) (2004) 329–366
33. Lamport, L.: The temporal logic of actions. ACM Transactions on Programming Languages and Systems (TOPLAS) **16**(3) (1994) 872–923
34. Merz, S.: Mechanizing tla in isabelle. In: Workshop on Verification in New Orientations, Citeseer (1995) 54–74
35. Grov, G., Merz, S.: A definitional encoding of tla* in isabelle/hol. Archive of Formal Proofs (November 2011) `http://isa-afp.org/entries/TLA.html`.
36. Merz, S.: A more complete tla. In: International Symposium on Formal Methods, Springer (1999) 1226–1244
37. Mattolini, R., Nesi, P.: An interval logic for real-time system specification. IEEE Transactions on Software Engineering **27**(3) (2001) 208–227
38. Schimpf, A., Merz, S., Smaus, J.G.: Construction of büchi automata for ltl model checking verified in isabelle/hol. In: International Conference on Theorem Proving in Higher Order Logics, Springer (2009) 424–439
39. Sickert, S.: Linear temporal logic. Archive of Formal Proofs (March 2016) `http://isa-afp.org/entries/LTL.html`.
40. Marmsoler, D.: Towards a theory of architectural styles. In: Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering - FSE 2014, ACM, ACM Press (2014) 823–825