FACTUM Studio: A Tool for the Axiomatic Specification and Verification of Architectural Design Patterns

Diego Marmsoler[0000−0003−2859−7673] and Habtom Kahsay Gidey

Technische Universität München

Abstract. Architectural Design Patterns (ADPs) restrict the design of an architecture with the aim to guarantee certain properties. Verifying ADPs requires to show that the imposed constraints indeed lead to the claimed guarantees and it is best done using interactive theorem proving (ITP). ITP, however, requires knowledge which is usually not available in the architecture community, which is why the technology is rarely used for the verification of patterns. To address this problem, we are working on a tool which supports the interactive verification of ADPs at a level of abstraction familiar to an architect. In the following paper, we introduce the tool and demonstrate it by means of a running example: we model a version of the Publisher-Subscriber pattern with a corresponding guarantee and verify it in a generated Isabelle/HOL theory.

Keywords: Architectural Design Patterns, Interactive Theorem Proving, FACTUM, Eclipse/EMF

1 Introduction

Architectural design patterns (ADPs) 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 [16]. They capture architectural design experience and provide abstract solutions to recurring architectural design problems. Thereby, they constrain an architectural design and guarantee some desired safety/liveness properties for architectures implementing the pattern. Thus, verifying ADPs requires to show that the constraints imposed by an ADP indeed lead to architectures satisfying the claimed guarantees. Due to the abstract nature of patterns, verification requires axiomatic reasoning and is usually done using interactive theorem proving (ITP) [12]. ITP, however, has a steep learning curve and requires expertise which is not always available in the architecture community. Thus, the potential of the technology is not yet fully explored for the verification of ADPs.

In an effort to lower the entry barriers and make the technology available to the broader software architecture community, we are developing FACTUM Studio, a tool which supports the interactive verification of ADPs at a level of abstraction familiar to an architect. Currently, its main features include support for the modeling of ADPs and the generation of corresponding Isabelle/HOL [15] theories. Thereby, it implements several languages, graphical as well as textual, to model different aspects of an ADP: abstract datatypes, component types, architectural constraints, and architectural guarantees. In this paper, we introduce FACTUM Studio: We describe its main features and demonstrate it in terms of a small, running example. The tool itself, as well as a step-by-step tutorial to reconstruct the running example, is provided online [7].

2 Overview of Core Features

Currently, FACTUM Studio provides the following core features to support the axiomatic specification of ADPs:

– Specification of abstract datatypes.
– Graphical modeling of component types.
– Specification of architectural constraints.
– Specification of architectural guarantees.

Thereby, FACTUM Studio provides rigorous type-checking mechanisms to support a user in the specification. Moreover, the tool allows to automatically generate a corresponding Isabelle/HOL theory from a given specification according to the algorithm presented in [12]. In the following, we describe each of the features in more detail and demonstrate them by means of a running example [1].

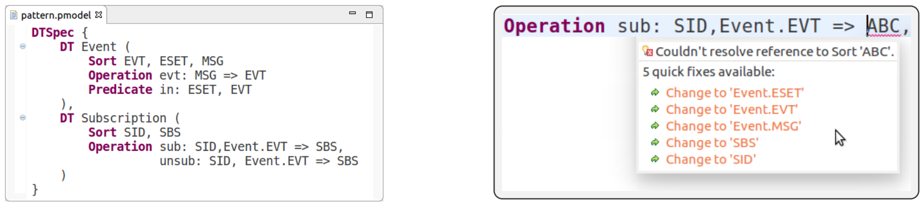
Example 1 (Running example: Publisher-Subscriber architectures). The purpose of the Publisher-Subscriber pattern is to achieve a “flexible” way of communication between components of an architecture. Thereby, the pattern requires the existence of two types of components: *publisher* and *subscriber* components. Subscribers can register for certain events by sending special subscription messages to publisher components. Publishers, on the other hand, can send out messages which are associated to certain events. Moreover, the pattern requires that a subscriber component is connected to a publisher, whenever the latter sends out a message for which the former previously subscribed. In return, the pattern guarantees for an architecture which satisfies these requirements, that a subscriber indeed receives all the messages associated to events for which he is subscribed. □

3 Specifying Data Types

Data types in FACTUM Studio are specified using traditional, *algebraic specification* techniques [3,17]. Therefore, for each datatype specification, we first provide a list of sorts. Then, we may use the sorts to specify characteristic functions for a data type. Finally, we may add axioms to characterize the behavior of the different functions. FACTUM Studio supports the specification of data types by means of two features: (i) First, it ensures that sorts used in a function specification do indeed exist. (ii) Moreover, it ensures that the parameters provided to a function in the specification of the axioms are consistent with the function’s signature.

Example 2 (Datatypes for Publisher-Subscriber architectures). The left-hand side of Fig. 1 depicts the specification of data types for the Publisher-Subscriber example in FACTUM Studio. It contains two datatype specifications: `Event` and `Subscription`.

[1] Note that the example is intentionally kept simple since its purpose is to demonstrate the tool’s main features, rather than evaluating it in a real-world setting.



```
DTSpec {
    DT Event (
        Sort EVT, ESET, MSG
        Operation evt: MSG => EVT
        Predicate in: ESET, EVT
    ),
    DT Subscription (
        Sort SID, SBS
        Operation sub: SID,Event.EVT => SBS,
                  unsub: SID, Event.EVT => SBS
    )
}
```

```
Operation sub: SID,Event.EVT => ABC,
Couldn't resolve reference to Sort 'ABC'.
5 quick fixes available:
Change to 'Event.ESET'
Change to 'Event.EVT'
Change to 'Event.MSG'
Change to 'SBS'
Change to 'SID'
```

Fig. 1. Datatype specification for Publisher-Subscriber architectures.

Specification `Event` contains the events (and sets of events) for which subscribers may subscribe as well as corresponding messages. Moreover, it declares a function `evt` to associate messages with events, and a predicate `in` to check whether an event is indeed contained within a set of events. Specification *Subscription*, on the other hand, contains identifiers `SID` for subscriber components and the actual subscriptions `SBS`. Moreover, it declares corresponding functions, `sub` and `unsub`, to subscribe to, and unsubscribe from events.

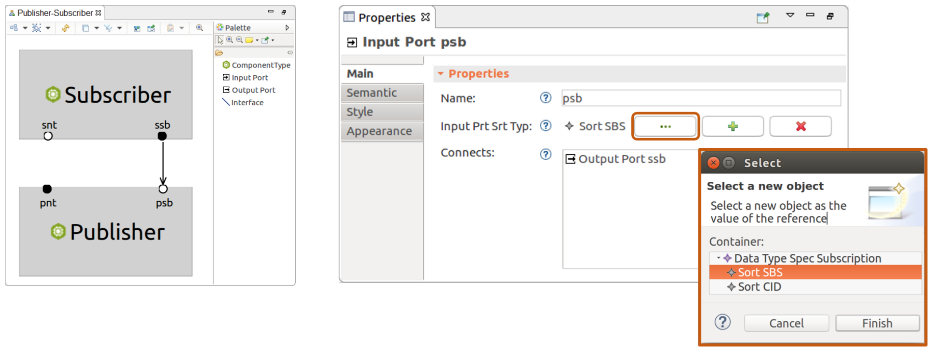
The right-hand side of Fig. 1 demonstrates a situation in which we used a *non-existent* sort `ABC` to specify operation `sub`. As shown in the figure, FACTUM Studio notifies the user about such erroneous situations. Moreover, it even suggests a list of existing sorts which may be used to fix the problem. □

Note that for our simple example, we did not even require to characterize the meaning of our declared functions and thus we did not specify any axioms for them. In more complicated examples, however, we may also specify axioms for datatype functions, which can then be used for the verification of the pattern. FACTUM Studio would support such specifications and ensure that the parameters provided to a function reference do indeed match the function's signature.

4 Modeling Interfaces

Interfaces for components are modeled in FACTUM Studio using so-called *architecture diagrams* [12]. Thereby, an interface is represented as a labeled rectangle with empty and filled circles, denoting input and output ports, respectively. Ports are typed by the datatypes introduced for the pattern by assigning a corresponding sort from the pattern's datatype specification to each port. In addition to specify interfaces, architecture diagrams may also be used to graphically express common architectural constraints. To this end, interfaces may be annotated with constraints about the activation of components. Moreover, connections between interface ports denote constraints about the connections between the corresponding ports of components.

Example 3 (Architecture diagram for Publisher-Subscriber architectures). Figure 2 depicts the specification of interfaces in FACTUM Studio for the Publisher-Subscriber example. The left-hand side shows the corresponding architecture diagram: It consists of two interfaces, `Publisher` and `Subscriber`, with one corresponding input and output port for both of them. Input port `psb` of the `Publisher` interface and output



Input Port psb

Name: psb

Input Prt Srt Typ: Sort SBS

Connects: Output Port ssb

Select a new object

Select a new object as the value of the reference

Container:

- Data Type Spec Subscription
  - Sort SBS
  - Sort CID

Fig. 2. Architecture diagram for Publisher-Subscriber architectures.

port `ssb` of the `Subscriber` interface are both typed by sort `EVT` introduced in Ex. 2. Moreover, the `Publisher`'s output port `pnt` and the `Subscriber`'s input port `snt` are both typed by sort `MSG`. The connection between ports `ssb` and `psb` specifies a constraint regarding the connection of corresponding ports for publisher and subscriber components, and it is discussed in more detail in the next section.

The right-hand side of Fig. 2 depicts the property panel for the publisher's input port `psb`: it contains its name and the names of connected ports, and allows to set the type of the port by assigning a corresponding sort from the pattern's datatype specification. □

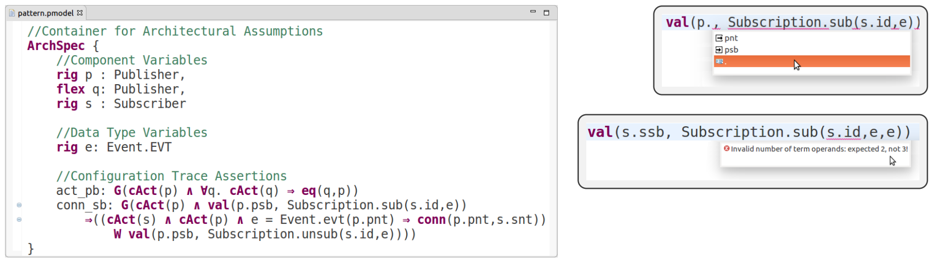
5 Adding Architectural Constraints

As discussed in [9,13], there exist 4 types of constraints for ADPs: (i) constraints for datatypes, (ii) constraints about the behavior of components, (iii) constraints about component activation, and (iv) connection between component ports. FACTum Studio implements languages for the specification of all of these types of constraints and provides rigorous type checking to support a user in their specification. Constraints for data types are specified in terms of axioms for the datatypes functions as already discussed in Sec. 3.

Constraints about component behavior are specified for interfaces in terms of so-called *behavior trace assertions* [10]: linear temporal logic formulæ with terms formulated by port names and datatype functions. By adding a behavior trace assertion to an interface, we obtain a corresponding *component type*, i.e., an interface with assertions about the behavior of corresponding components.

Constraints about component activation and connections between ports are expressed by means of so-called *architecture trace assertions*: linear temporal logic formulæ with special predicates to denote component activation and port connection. Compared to behavior trace assertions (which are specified over a single interface), architecture trace assertions are specified over an architecture (a set of components). Thus, their specification relies on the concept of *component variables*: variables which are interpreted by components of a certain type.

Example 4 (Architectural constraints for Publisher-Subscriber architectures). The left-hand side of Fig. 3 depicts the specification of two architectural constraints for Publisher-



```
//Container for Architectural Assumptions
ArchSpec {
    //Component Variables
    rig p : Publisher,
    flex q: Publisher,
    rig s : Subscriber

    //Data Type Variables
    rig e: Event.EVT

    //Configuration Trace Assertions
    act_pb: G(cAct(p) ∧ ∀q. cAct(q) ⇒ eq(q,p))
    conn_sb: G(cAct(p) ∧ val(p.psb, Subscription.sub(s.id,e))
        ⇒((cAct(s) ∧ cAct(p) ∧ e = Event.evt(p.pnt) ⇒ conn(p.pnt,s.snt))
            W val(p.psb, Subscription.unsub(s.id,e))))
}
```

```
val(p., Subscription.sub(s.id,e))
```

```
val(s.ssb, Subscription.sub(s.id,e,e))
```

Invalid number of term operands: expected 2, not 3!

Fig. 3. Architectural constraints for Publisher-Subscriber architectures.

Subscriber architectures in FACTUM Studio: **act_pb** requires that a publisher component is always activated and *unique* and **conn_sb** requires that a subscriber component is connected to the unique publisher component, whenever the latter sends out a message for which the subscriber is subscribed. To specify the constraints, we first declared three component variables: a *rigid*[2] one p and a *flexible*[3] one q for publisher components as well as a rigid one s for subscriber components. In addition, we declared one rigid datatype variable e for events, i.e., messages of type `EVT`. Then, we specified the two constraints in terms of two linear temporal logic formulæ over these variables. Therefore, we use four, so-called, *architecture predicates* to denote i. valuation of component ports with certain messages (`val`), ii. component activation (`cAct`), iii. connection between component ports (`conn`), and iv. equality of components (`eq`).

The right-hand side of Fig. 3 demonstrates some of FACTUM Studio's features to support the specification of architectural constraints. One convenient feature, which is demonstrated at the top of the figure, is its support to specify component ports: whenever such a port is used in the specification, the tool ensures that the port is indeed consistent with the type of the corresponding component variable. Thus, since variable s is of type `Subscriber`, only ports `snt` and `ssb`, declared for a subscriber interface, can be referenced. Another convenient feature is the signature check, as demonstrated at the bottom of the figure: whenever we use a datatype function, FACTUM Studio ensures that the parameters passed to the function (either port values or other functions) are indeed consistent with the function's signature (as specified in the corresponding datatype specification). Thus, since function `sub` was declared to take two parameters only (see Sec. 2), we are indeed not allowed to pass more than two parameter to `sub`.

6 Specifying Architectural Guarantees

Architectural guarantees are specified similar to activation or connection constraints using *architecture trace assertions* (as introduced in Sec. 5). Consequently, FACTUM Studio supports its specification with the same features as described in Sec. 5 for the specification of architectural constraints.

[2] A rigid variable keeps its value over time.

[3] A flexible variable is newly interpreted at each point in time.

Example 5 (A guarantee for Publisher-Subscriber architectures). Figure 4 depicts the specification of an architectural guarantee **delivery** for Publisher-Subscriber architectures in FACTUM Studio. It ensures that a subscriber indeed receives all the messages for which he is subscribed and it is specified in a similar manner as the architectural assumptions discussed in Ex. 4 using component variables and architecture predicates. As mentioned before, FACTUM Studio provides similar support for the specification of architectural guarantees as it does for architectural assumptions: correct use of component variables and architecture predicates as well as type checking for datatype functions and ports.

7 Verifying the Pattern

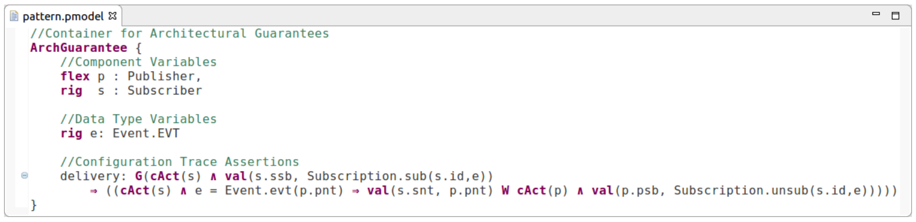
After specifying a pattern in FACTUM Studio, we can generate corresponding Isabelle/HOL theories. The theory consists of an Isabelle/HOL locale [2] specification, which contains corresponding assumptions for each architectural constraint. Moreover, the pattern's guarantee is used to generate a corresponding Isabelle/HOL theorem for the locale which can then be proved using our calculus for dynamic architectures [11].

Example 6 (Mapping Publisher-Subscriber architectures). Using FACTUM Studio, we can create an Isabelle/HOL theory for the Publisher-Subscriber pattern containing assumptions for all the architectural constraints specified so far. Moreover, the theory contains also a theorem derived from the corresponding architectural guarantee. The generated theorem can be easily proved from the assumption using Isabelle/Isar [4].

8 Related Work

In the following, we present reviews of related tools that provide support for software architecture specification and verification. For instance, The Alloy Analyzer [8] is an automated modeling and analysis tool, based on a declarative language named Alloy. It supports the specification of constraints which can then be transformed to a corresponding specification for the Boolean SAT Solver Kodkod for analysis. Acme Studio [6] is a tool which focuses on the structural representations of architectural designs

[4] A corresponding proof is provided online in this paper's supplementary electronic material.



```
//Container for Architectural Guarantees
ArchGuarantee {
    //Component Variables
    flex p : Publisher,
    rig  s : Subscriber

    //Data Type Variables
    rig e: Event.EVT

    //Configuration Trace Assertions
    delivery: G(cAct(s) ∧ val(s.ssb, Subscription.sub(s.id,e))
        ⇒ ((cAct(s) ∧ e = Event.evt(p.pnt) ⇒ val(s.snt, p.pnt) W cAct(p) ∧ val(p.psb, Subscription.unsub(s.id,e)))))
}
```

Fig. 4. A guarantee for Publisher-Subscriber architectures.

and which provides an editor and visualization environment. It supports analysis of specifications by means of static checks using the Armani constraint checker. ArchStudio [5] provides an Eclipse-based environment for architecture specifications. It uses the Schematron XML constraint engine for static analyses. RoboCalc [14] supports the modeling and analysis of autonomous mobile robots, based on an UML-like notation called RoboChart. It supports analyses of such models using model checking. AutoFocus [1] supports the development process of software-intensive, embedded systems, starting from requirements to code generation. It allows translating models to the NuSMV/nuXmv model checker for analysis. VerCors (VCE) [4] is a platform to support the graphical specification of architectures based on the Grid Component Model (GCM). It supports the analysis of such models through model checking and equivalence checking.

While all these tools can be used to specify and analyze architectures or even architectural constraints, they focus mainly on automatic analyses and thus lack the support for component types. With the work presented in this paper, we complement these approaches by providing an alternative approach to architecture analysis, based on axiomatic specification techniques and interactive theorem proving.

9 Conclusion

In this paper, we introduced FACTUM Studio, a tool to support the axiomatic specification and verification of architectural design patterns (ADPs). The tool allows to specify an ADP using a combination of graphical and textual elements and supports the specification by various type checking mechanisms. Then, a corresponding Isabelle/HOL theory can be generated out of the specification.

Our overall goal is the development of tools to support the interactive verification of ADPs. With the work presented in this paper, we provide an important, first step towards this goal. There is, however, still some missing elements to fully achieve this goal, which leads to future work: (i) The specification language should be extended to deal with more advanced features, such as parametric component types and hierarchical specifications. (ii) The output of Isabelle/HOL should be interpreted back into the modeling environment. (iii) In the long term, FACTUM Studio should also support modeling of proofs at the architecture level.

Acknowledgments: We would like to thank Dominik Ascher and Sebastian Wilzbach for their valuable support on Eclipse/EMF. Parts of the work on which we report in this paper was funded by the German Federal Ministry of Economics and Technology (BMWi) under grant no. 0325811A.

References

1. Aravantinos, V., Voss, S., Teufl, S., Hölzl, F., Schätz, B.: Autofocus 3: Tooling concepts for seamless, model-based development of embedded systems. In: ACES-MB&WUCOR@MoDELS. pp. 19–26 (2015)
2. Ballarin, C.: Locales and locale expressions in isabelle/isar. Lecture notes in computer science 3085, 34–50 (2004)

3. Broy, M.: Algebraic specification of reactive systems. In: Algebraic Methodology and Software Technology. pp. 487–503. Springer, Springer Berlin Heidelberg (1996)
4. Cansado, A., Madelaine, E., Valenzuela, P.: Vce: A graphical tool for architectural definitions of gcm components. In: 5th workshop on Formal Aspects of Component Systems (FACS'08) (2008)
5. Dashofy, E.M.: Supporting stakeholder-driven, multi-view software architecture modeling. Ph.D. thesis, University of California, Irvine (2007)
6. Garlan, D., Monroe, R.T., Wile, D.: Acme: Architectural description of component-based systems. Foundations of component-based systems 68, 47–68 (2000)
7. Gidey, H.K., Marmsoler, D.: FACTUM Studio. `https://habtom.github.io/factum/` (2018)
8. Jackson, D.: Alloy: a lightweight object modelling notation. ACM Transactions on Software Engineering and Methodology (TOSEM) 11(2), 256–290 (2002)
9. Marmsoler, D., Gleirscher, M.: On activation, connection, and behavior in dynamic architectures. Scientific Annals of Computer Science 26(2), 187–248 (2016)
10. 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)
11. Marmsoler, D.: Towards a calculus for dynamic architectures. In: Hung, D.V., Kapur, D. (eds.) Theoretical Aspects of Computing - ICTAC 2017 - 14th International Colloquium, Hanoi, Vietnam, October 23-27, 2017, Proceedings. Lecture Notes in Computer Science, vol. 10580, pp. 79–99. Springer (2017), `https://doi.org/10.1007/978-3-319-67729-3_6`
12. Marmsoler, D.: Hierarchical specication and verication of architecture design patterns. In: Fundamental Approaches to Software Engineering - 21th International Conference, FASE 2018, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2018, Thessaloniki, Greece, April 14-20, 2018, Proceedings (2018)
13. Marmsoler, D., Gleirscher, M.: Specifying properties of dynamic architectures using configuration traces. In: International Colloquium on Theoretical Aspects of Computing, pp. 235–254. Springer (2016)
14. Miyazawa, A., Cavalcanti, A., Ribeiro, P., Li, W., Woodcock, J., Timmis, J.: Robochart reference manual. Tech. rep., Technical report, University of York (2017)
15. Nipkow, T., Paulson, L.C., Wenzel, M.: Isabelle/HOL: a proof assistant for higher-order logic, vol. 2283. Springer Science & Business Media (2002)
16. Taylor, R.N., Medvidovic, N., Dashofy, E.M.: Software Architecture: Foundations, Theory, and Practice. Wiley Publishing (2009)
17. Wirsing, M.: Algebraic specification. In: van Leeuwen, J. (ed.) Handbook of Theoretical Computer Science (Vol. B), pp. 675–788. MIT Press, Cambridge, MA, USA (1990), `http://dl.acm.org/citation.cfm?id=114891.114904`