Modeling and Verifying Dynamic Architectures with FACTum Studio

Habtom Kahsay Gidey[1], Alexander Collins[2], and Diego Marmsoler[2]

[1] Universität der Bundeswehr München
[2] Technische Universität München

Abstract. With the emergence of ambient and adaptive computing, dynamic architectures have become increasingly important. Dynamic architectures describe an evolving state space of systems over time. In such architectures, components can appear or disappear, and connections between them can change over time. Due to the evolving state space of such architectures, verification is challenging. To address this problem, we developed FACTum Studio, a tool that combines model checking and interactive theorem proving to support the verification of dynamic architectures. To this end, a dynamic architecture is first specified in terms of component types and architecture configurations. Next, each component type is verified against asserted contracts using nuXmv. Then, the composition of the contracts is verified using Isabelle/HOL. In this paper, we discuss the tool's extended features with an example of an encrypted messaging system. It is developed with Eclipse and active on Github.

Keywords: Dynamic Architectures, Model Checking, Interactive Theorem Proving, FACTUM, Eclipse/EMF, Xtext

1 Introduction

Software systems that enact self-adaptation, learning, and complex reasoning are inherently dynamic [7,9,21]. They autonomously change their structure and composition at run time [5,20]. Such systems can be specified in two steps: First, a set of component types is specified using state machines [18,10]. Next, the composition of components is specified using architectural assertions [4,13].

The dynamic nature of such systems leads to a dynamically evolving state space, which makes their verification difficult. Sometimes, we lack even upper bounds on the active number of components, and so verification requires reasoning over an unbounded state space. In FACTUM [14,17], we address this problem by splitting the verification process into two steps. To verify a property for the overall architecture, we first identify suitable contracts for the component types and verify them against their implementation using model checking. In a second step, we apply interactive theorem proving (ITP) to combine these verified contracts to derive the overall system property. The restricted state space of single component types enables automatic verification techniques and allows fast feedback on the satisfaction of contracts. On the other hand, the expressiveness of

ITP allows us to reason about potentially unbounded numbers of components when verifying overall system properties.

In a previous work [16], we presented a preliminary version of FACTum Studio, an Eclipse-based modeling application supporting the specification of Architectural Design Patterns and their verification in Isabelle/HOL. With this paper, we introduce an extended version, adding the following features: (i) Support for graphical modeling of component behavior using annotated state machines. (ii) Corresponding code generation for the nuXmv model checker.

2 Background

We approach the development of FACTum Studio following the FACTum framework [17,14]. Fig. 1 illustrates the essential parts of the implemented framework. It depicts the process of architecture specification, including the creation of theories and models for verification. It also shows where ITP and model checking tools are utilized.

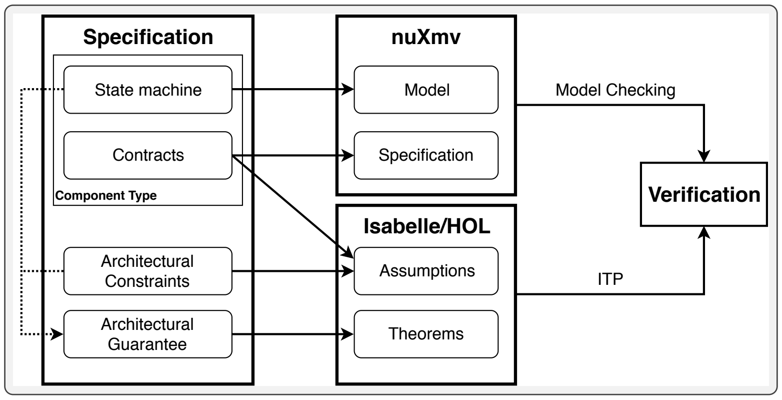


Fig. 1: Verifying Dynamic Architectures in FACTum Studio.

In our previous paper [16] we reported the first release of FACTum Studio. We discussed its basic features, introducing the graphical and textual language that supports architectural pattern specifications. These core features facilitate the specification and representation of abstract data types, component types, and architectural assertions in terms of constraints and guarantees.

3 FACTum Studio 2

The second release of FACTum Studio adds the following new features: (i) textual and graphical modeling of behavior for components in terms of state machines, (ii) specification of contracts for component types in terms of LTL-formulæ, (iii) generation of models for the nuXmv model checker from component behaviors, (iv) generation of specifications for nuXmv from component type contracts, and (v) generation of additional assumptions for Isabelle locales from contracts. Moreover, the new release contains additional feature enhancements and bug fixes.

Example 1 (Secure Messaging System). Fig. 2 shows an architectural diagram consisting of the key elements of the messaging protocol. The architecture represents a simple encrypted message exchange system between a sender and a receiver. Messages are encrypted with a key by the sender and decrypted with the same key on the receiving end. However, the nodes are restricted to forwarding the messages and cannot read messages in the middle. The `Encrypt` component type has an input port and output port which connects to a `Node` component type. The `Node` component type also contains ports which connect it to the `Encrypt` and `Decrypt` component types. Similarly, the `Decrypt` component type has ports connecting it to a `Node` component type. The `Node` component type represents an arbitrary number of dynamic nodes. They can join and leave the network at any time dynamically. A message moves through the active nodes until it reaches its destination. The full textual specification of the running example is provided in the project repository in GitHub [8].

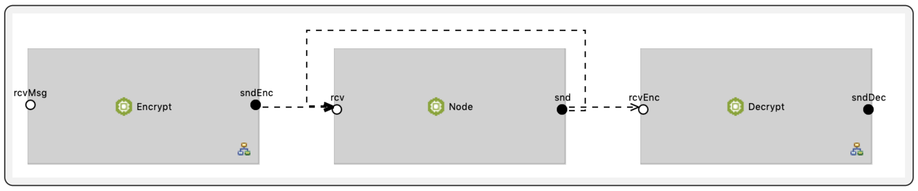


Fig. 2: Running Example - Secure Messaging, Architecture Diagram.

4 Specification of Dynamic Architectures

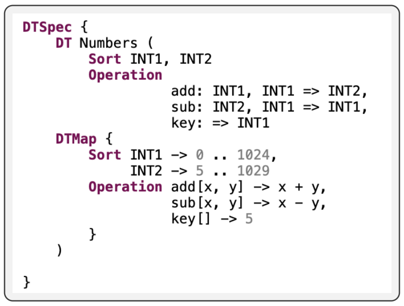
In FACTum, specifications usually begin by describing essential architectural elements such as data types and component types. In the following, we use text boxes with a light bulb icon to highlight the new features with FACTum Studio 2.

4.1 Specifying Datatypes

In FACTum Studio, data types are specified using algebraic specification techniques [22,4].

In FACTum Studio 2 we extended the specification of data types by allowing the user to map FACTum sorts to nuXmv data types and FACTum operations to nuXmv operations.

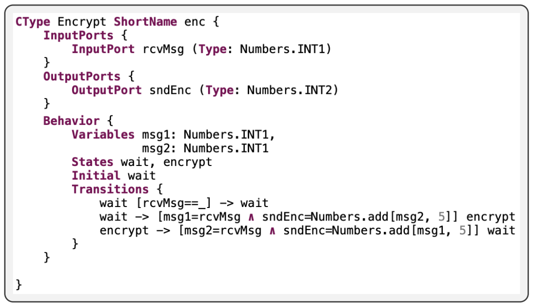
Example 2 (Data Types for Secure Messaging System). Fig. 3 shows a data type specification for our example. It declares a data type `Numbers` and its mapping to `nuXmv` data type defined in `DTMap`. The specification of data types' operations is enhanced in the new release to include several possibilities, such as ranges of numbers.



```
DTSpec {
    DT Numbers (
        Sort INT1, INT2
        Operation
                  add: INT1, INT1 => INT2,
                  sub: INT2, INT1 => INT1,
                  key: => INT1
    DTMap {
        Sort INT1 -> 0 .. 1024,
             INT2 -> 5 .. 1029
        Operation add[x, y] -> x + y,
                  sub[x, y] -> x - y,
                  key[] -> 5
        }
    )

}
```

Fig. 3: Datatype Specification



```
CType Encrypt ShortName enc {
    InputPorts {
        InputPort rcvMsg (Type: Numbers.INT1)
    }
    OutputPorts {
        OutputPort sndEnc (Type: Numbers.INT2)
    }
    Behavior {
        Variables msg1: Numbers.INT1,
                  msg2: Numbers.INT1
        States wait, encrypt
        Initial wait
        Transitions {
            wait [rcvMsg==_] -> wait
            wait -> [msg1=rcvMsg ∧ sndEnc=Numbers.add[msg2, 5]] encrypt
            encrypt -> [msg2=rcvMsg ∧ sndEnc=Numbers.add[msg1, 5]] wait
        }
    }

}
```

Fig. 4: Component Type Specification

4.2 Component Types

Component types are specified using architecture diagrams. Figure 2, for example, shows an architecture diagram for our running example.

FACTum Studio 2 adds support for the textual or graphical behavior specification of component types using state machines.

Example 3 (Behavior for Encrypt Component Type). In Fig. 4 we show an example of the textual description of the `Encrypt` component type. It describes the component's two ports, the input port `rcvMsg` and the output port `sndEnc`. The `Behaviour` code block describes the specification of the component type's behavior. It contains a description of states, `wait` and `encrypt`. It defines an initial state `wait` where its initial value is set to `rcvMsg`. It also defines the `wait` and `encrypt` state transitions as a start and end of the transition behavior, respectively. Fig. 5 shows the behavior specified graphically. The graphical representations are also editable with text annotations.

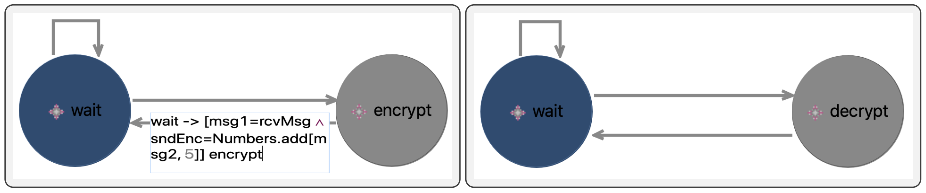
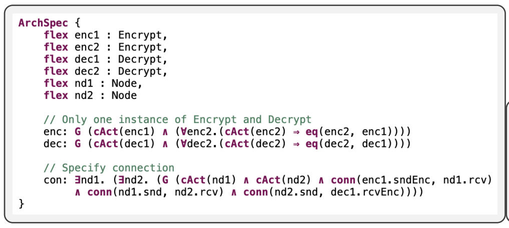


Fig. 5: Graphical Behavior Specification with State Machines.

4.3 Architecture Specification

In FACTUM Studio, architectural configurations are specified using architectural assertions: LTL-formulæ expressed over the architecture. In FACTUM Studio 2, the architecture specification did not change much [16].

Example 4 (Architecture Specification for Secure Messaging System). In Fig. 6, we have the architectural constraints set over sets of component instances. The formulæ `enc` and `dec` assert a property that requires a unique decryption and a unique encryption components set active at any time point. The `con` predicate specifies the connection property of components within the architecture.

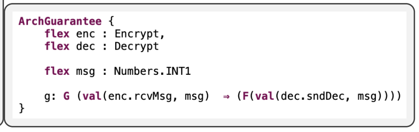


```
ArchSpec {
    flex enc1 : Encrypt,
    flex enc2 : Encrypt,
    flex dec1 : Decrypt,
    flex dec2 : Decrypt,
    flex nd1 : Node,
    flex nd2 : Node

    // Only one instance of Encrypt and Decrypt
    enc: G (cAct(enc1) ∧ (∀enc2.(cAct(enc2) ⇒ eq(enc2, enc1))))
    dec: G (cAct(dec1) ∧ (∀dec2.(cAct(dec2) ⇒ eq(dec2, dec1))))

    // Specify connection
    con: ∃nd1. (∃nd2. (G (cAct(nd1) ∧ cAct(nd2) ∧ conn(enc1.sndEnc, nd1.rcv)
         ∧ conn(nd1.snd, nd2.rcv) ∧ conn(nd2.snd, dec1.rcvEnc))))
}
```

Fig. 6: Architectural Constraints



```
ArchGuarantee {
    flex enc : Encrypt,
    flex dec : Decrypt

    flex msg : Numbers.INT1

    g: G (val(enc.rcvMsg, msg)  ⇒ (F(val(dec.sndDec, msg))))
}
```

Fig. 7: Architectural Guarantees

5 Verification using Model Checking and ITP

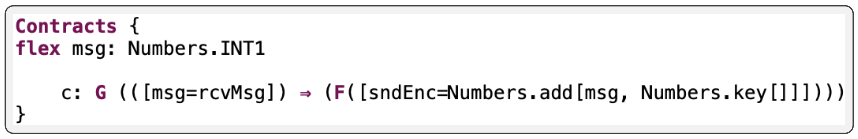
In FACTUM, architectures are verified in two steps: First, we specify the architectural guarantees and establish suitable contracts. Contracts are then verified against described behaviors of component types using model checking. Secondly, the verified contracts are combined with the architectural specifications to verify the overall architecture using interactive theorem proving.

Example 5 (Guarantee for Secure Messaging System). In Fig. 7, the architectural guarantee `g` ensures a property that when a message `msg` is sent by the sender it will eventually be delivered to the receiver and cannot be read in between.

FACTUM Studio 2 now supports the specification of contracts for component types using LTL-formulæ over their ports.

5.1 Verifying Component Types

As discussed in Sec. 1, FACTUM Studio automatically verifies individual implementations of component types using the model checker nuXmv. To this end, we first identify suitable contracts for each type of component. Contracts specify behaviors of component types expressed as behavior trace assertions [15].



```
Contracts {
flex msg: Numbers.INT1

    c: G (([msg=rcvMsg]) ⇒ (F([sndEnc=Numbers.add[msg, Numbers.key[]]])))
}
```

Fig. 8: Contract for the Encrypt Component Type.

Example 6 (Contracts for component types). In Fig. 7, a suitable contract for the encryption component type is specified as `c`. The contract describes the desired property for encrypting and forwarding every message. It asserts a liveness property so that, eventually, every input is encrypted.

Then, FACTum Studio can be used to generate nuXmv code for each component type. The corresponding algorithm is shown in Algorithm 1. It describes the mapping process followed during the model transformations.

FACTum Studio 2 now supports the generation of nuXmv code.

Algorithm 1 nuXmv code generation

```
Require: pattern
 1: for all component types do
 2:   for all states do
 3:     add state to enum
 4:   end for
 5:   for all state machine variables do
 6:     encode as variable with sort
 7:   end for
 8:   for all input ports and output ports do
 9:     encode as variable with sort
10:     create noVal port as boolean
11:   end for
12:   encode initialization of variables
13:   encode state machines in assignment style
14:   encode transitions of noVal output ports in assignment style
15:   for all contracts do
16:     encode as LTLSpec
17:   end for
18: end for
```

5.2 Verifying the Architecture

As a final step, FACTum Studio generates Isabelle/HOL and nuXmv code for architecture verification. It transforms the specified architectural assertions to corresponding Isabelle/HOL assumptions and theories. Similarly, the state machines and contracts of individual components are transformed into corresponding nuXmv models and specifications, see Fig. 1. At this stage, FACTum Studio checks the state machine models of individual component types against their contracts. Next, it combines the verified contracts with the Isabelle/HOL theorems generated from the architectural guarantee. Using ITP, we can then sketch

Isabelle/HOL proof interactively to finalize the verification of dynamic architectures.

FACTUM Studio 2 now also exports component contracts as assumptions for the corresponding Isabelle locale.

Example 7. The complete generated Isabelle/HOL theory and the proof for the running example are provided in the code repository [8].

6 Related Work

The first category and a significant number of related works are tools such as AutoFocus, RoboChart, VerCor, and others using languages DynAlloy, Dynamic Wright [1,6,19]. These tools use automated verification with model checking or an analyzer such as the Alloy Analyzer. The tools in this category do not provide support for ITP based dynamic architecture verification.

The second category of tools provides support for model transformations of architecture specifications to proof assistants. These tools are very few in number and would include the previous version of FACTUM Studio, Reo, and STeP [2,3,11,12,16].

However, to the best of our knowledge, the new FACTUM Studio is the only tool to use a combined approach of both ITP and model checking for the verification of dynamically adapting, component-based systems.

7 Conclusion and Outlook

Verification of dynamic architecture specifications for software-intensive systems requires a combined approach to address the varying nature of challenges in the domain. In this paper, with the implementation of new features in FACTUM Studio, we have described how to approach and address the state-space scalability problem of dynamic architecture verification by using model checking and ITP. First, we demonstrated how individual architectural elements and their dynamic properties are specified and verified using a model checker. We then used individually checked contracts to verify assured compositional constraints using ITP.

Since ITP requires a steep learning curve, in the next versions of FACTUM Studio, we plan to implement some form of automatic proof generation from specified guarantees. Moreover, we also plan to support the hierarchical specification of component types and their configurations. That can enable users to import, use, or extend already specified and verified dynamic architecture patterns.

Acknowledgments: Parts of the work on which we report in this paper were 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: CEUR Workshop Proceedings. vol. 1508, pp. 19–26. CEUR-WS.org (2015)
2. Arbab, F.: Reo: a channel-based coordination model for component composition. Mathematical structures in computer science 14(03), 329–366 (2004)
3. Baier, C., Sirjani, M., Arbab, F., Rutten, J.: Modeling component connectors in reo by constraint automata. Science of computer programming 61(2), 75–113 (2006)
4. Broy, M.: A model of dynamic systems. In: Bensalem, S., Lakhneck, Y., Legay, A. (eds.) From Programs to Systems. The Systems Perspective in Computing, Lecture Notes in Computer Science, vol. 8415, pp. 39–53. Springer Berlin Heidelberg (2014)
5. Bruni, R., Bucchiarone, A., Gnesi, S., Melgratti, H.: Modelling dynamic software architectures using typed graph grammars. Electronic Notes in Theoretical Computer Science 213(1), 39–53 (2008)
6. Bucchiarone, A., Galeotti, J.P.: Dynamic software architectures verification using DynAlloy. Electronic Communications of the EASST 10 (2008)
7. Gerostathopoulos, I., Skoda, D., Plasil, F., Bures, T., Knauss, A.: Architectural homeostasis in self-adaptive software-intensive cyber-physical systems. In: European Conference on Software Architecture. pp. 113–128. Springer (2016)
8. Gidey, H.K., Marmsoler, D.: FACTum Studio. https://habtom.github.io/factum/ (2018)
9. Koza, J.R.: Genetic programming: on the programming of computers by means of natural selection, vol. 1. MIT press (1992)
10. Li, C., Huang, L., Chen, L., Li, X., Luo, W.: Dynamic software architectures: formal specification and verification with csp. In: Proceedings of the Fourth Asia-Pacific Symposium on Internetware. p. 5. ACM (2012)
11. 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, China. Lecture Notes in Computer Science, vol. 8348, pp. 273–290. Springer (2013), https://doi.org/10.1007/978-3-319-07602-7_17
12. Manna, Z., Sipma, H.B.: Deductive verification of hybrid systems using STeP. In: International Workshop on Hybrid Systems: Computation and Control. pp. 305–318. Springer (1998)
13. Marmsoler, D.: Towards a calculus for dynamic architectures. In: Hung, D.V., Kapur, D. (eds.) Theoretical Aspects of Computing - ICTAC 2017 - 14th International Colloquium, Proceedings. Lecture Notes in Computer Science, vol. 10580, pp. 79–99. Springer (2017), https://doi.org/10.1007/978-3-319-67729-3
14. Marmsoler, D.: A framework for interactive verification of architectural design patterns in Isabelle/HOL. In: International Conference on Formal Engineering Methods. pp. 251–269. Springer (2018)
15. Marmsoler, D.: Hierarchical specification and verification of architectural 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, Proceedings. Lecture Notes in Computer Science, vol. 10802. Springer (2018), https://doi.org/10.1007/978-3-319-89363-1
16. Marmsoler, D., Gidey, H.K.: FACTum Studio: a tool for the axiomatic specification and verification of architectural design patterns. In: International Conference on Formal Aspects of Component Software. pp. 279–287. Springer (2018)

17. Marmsoler, D., Gidey, H.K.: Interactive verification of architectural design patterns in FACTum. Formal Aspects of Computing (2019)
18. Marmsoler, D., Gleirscher, M.: Specifying properties of dynamic architectures using configuration traces. In: International Colloquium on Theoretical Aspects of Computing, Lecture Notes in Computer Science, vol. 9965, pp. 235–254. Springer (2016)
19. Miyazawa, A., Cavalcanti, A., Ribeiro, P., Li, W., Woodcock, J., Timmis, J.: Robochart reference manual. Tech. rep., Technical report, University of York (2017)
20. Oquendo, F.: Dynamic software architectures: formally modelling structure and behaviour with Pi-ADL. In: 2008 The Third International Conference on Software Engineering advances. pp. 352–359. IEEE (2008)
21. Oreizy, P., Gorlick, M.M., Taylor, R.N., Heimhigner, D., Johnson, G., Medvidovic, N., Quilici, A., Rosenblum, D.S., Wolf, A.L.: An architecture-based approach to self-adaptive software. IEEE Intelligent Systems and Their Applications 14(3), 54–62 (1999)
22. 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`