Are Formal Contracts a useful Digital Twin of Software Systems?

Jonas Schiffl[1], Alexander Weigl[1,*,†]

[1]*Karlsruhe Institute of Technology, Karlsruhe, Germany*

Abstract

Digital Twins are a trend topic in the industry today to either manage runtime information or forecast properties of devices and products. The techniques for Digitial Twins are already employed in several disciplines of formal methods, in particular, formal verification, runtime verification and specification inference. In this paper, we connect the Digital Twin concept and existing research areas in the field of formal methods. We sketch how digital twins for software-centric systems can be forged from existing formal methods.

Keywords

Formal Verification, Runtime Verification, Specification Mining, Temporal Logics

1. Introduction

Motivation *Digital Twin* is an emerging trend in many industries. The main focus is on the coupling of cyber-physical systems to a digital representation of the system, called the Digital Twin. The shape of a Digital Twin is tailored to the twinned cyber-physical system and the required reasoning. However, boundary constraints, such as performance restrictions, can also be involved. For example, a Digital Twin for predictive maintenance, i.e., the forecast of upcoming failures due to attrition, has different model elements from a Digital Twin for tracking the material flow.

Formal methods are a well-established research area. Although niche in industry adaption, they offer a rich toolbox for modeling systems. Therefore, formal methods are a natural candidate for digital twins of software systems. A recent survey paper states that only three of the surveyed approaches for Digital Twins for industrial automation systems use formal methods in contrast to 13 papers in the category "Exploratory investigation" and 17 in category "Testing" ([1, Table 1]).

In this work, we show how formal methods, in particular formal contracts, can be used to build a Digital Twin of a software system, update it according to the state of the system, and derive predictions about system behavior and safety properties.

FMDT'23: Workshop on Applications of Formal Methods and Digital Twins, March 13, 2023, Lübeck, Germany

**Corresponding author.

jonas.schiffl@kit.edu (J. Schiffl); weigl@kit.edu (A. Weigl)

0000-0002-9882-8177 (J. Schiffl); 0000-0001-8446-4598 (A. Weigl)

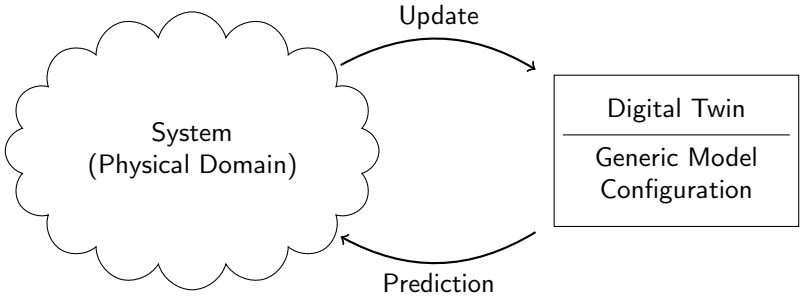

Figure 1: A Meta-model Digital Twin

Contribution In this paper, we apply several formal methods and tools to establish a framework for Digital Twins of reactive systems using formal software contracts. Therefore, we focus on twinning the software behavior of a reactive system operation according to its formal model, which we define through contracts.

The Digital Twin allows forecasting properties that depend on the behavior of the system. However, forecasting is not limited to the specific instance to which it is twinned. Moreover, knowledge learned from observation can be shared between Digital Twins, and What-if analyses are available. To achieve this, we show how existing formal methods, in particular formal verification, runtime verification, specification mining and program repair, can be combined for a Digital Twin.

A meta-model for Digital Twin For this paper, we assume a simple meta-model of Digital Twins, sketched in Figure 1. This meta-model consists of the original system that exists in the real world. Such systems can be cyber-physical systems, like cars, automated production systems, or energy systems. For the purposes of this discussion, we only consider pure software systems.

The Digital Twin is then a digital representation of an instance of a system. In case of a cyber-physical system, this can be a specific car or an individual transformer in the energy system. Coming from the domain of formal methods, we build the Digital Twin by using symbolic logic based models. We will later use contract-languages to represent the system. For software systems, the Digital Twin can be split into a generic part that is valid for all instances of the system parameters, and instance-specific configuration. For example, the generic consists of the classes and their contracts, but their use and composition is determined by the instance's configuration. Note that the Digital Twin also contains information about the environment. This is similar to the distinction between ABox and TBox in description logics. The TBox, the generic part, holds the axioms that are valid for all instances. The ABox, on the other hand, contains the information for the specific instance [2]. The TBox determines which parts of the ABox are relevant.

Operations The representation of a system as a Digital Twin should enable two operations:

- *Prediction* of selected properties. Prediction means reasoning whether a property holds for a specific instance or, due to the symbolic representation, for a family of instances. Moreover, we can perform "What-If" analyses: assuming a different configuration, we can predict properties on different instances, after change on the system. We can also model a transfer to a different environment. Prediction can lead to effects on the system, e.g., only configurations with a good predicted performance are deployed.
- The *Update* process consists of collecting and aggregating information from the (real-world) instance and transferring these into the configuration part of the model.

Desired Properties A Digital Twin also has to fulfill some properties. First, it should be faithful in the sense that the derived predictions are precise. If precision is too hard to achieve, we claim at least soundness: The prediction should be a pessimistic (or conservative) evaluation with respect to the safety analysis. The reasoning based on the Digital Twin should never attest safety falsely; if in doubt, it should rather predict non-existing safety issues.

Additionally, compositionality of a Digital Twin brings the same advantages as for the system. Systems can be built up by composition of multiple systems. The same is desirable for the Digital Twins: the Digital Twin of the top system is, ideally, a composition of the Digital Twins of the sub systems.

2. Contracts for Reactive Systems

In the following, we instantiate the Digital Twin meta-model for reactive software systems. For this, we define the individual components: the system and the formal model as well as the prediction and update processes.

2.1. The system.

A reactive system is characterized by the following properties (cf. [3]): Their runtime is infinite or undetermined, and they interact with the physical world, e.g., by reading sensor values and controlling actuators. Typical representatives for reactive systems are embedded controllers in the automation or automotive domain.

Reactive systems can be constructed via composition from smaller systems. For example, Lingua Franca [4] defines reactive systems, named *reactors*, that can be built of other reactors, or an executable program fragment. A similar terminology can be found in IEC 61131-3 (the standard for programming languages for automated production systems) as *Function Block Diagrams*. In this paper, we assume the system model of [5], in which every system has an interface (input and output variables) and is composed of connected subsystems.

2.2. The Digital Twin.

Cimatti et al. [5] provide OCRA, a tool that enables a design-by-contract methodology for reactive systems by defining components, their composition and associated contracts. OCRA focuses on the refinement relationship, thus verification of the program code against contract is left out.

Using the OCRA model, the Digital Twin of a system is its contract, which consists of two properties given in Linear Temporal Logic (LTL). The first property is an assumption on the system inputs. The second one is a guarantee of the system outputs under the given assumptions. Note that LTL is just the language we use for describing sets of traces. It can be replaced by any logic on traces (see below). The advantage of LTL is acceptance by a wide range of tools.

In addition to the contract, the Digital Twin is also aware of the structural architecture of the software given by the composition of subsystems and the information flow between them. Parameterization (or configuration) of a system is established via its input variables and by the selection of the sub-system during the composition.

The contract is a syntactical notion that describe the set of allowed behaviors of a system under a specified set of input traces. The behavior of a correct system is a subset of the allowed behavior. Often, due to the determinism in the system and the indeterminism (or abstraction) in the contract, the allowed behavior is indeed a strict superset of the actual actual behavior. Additionally, a system composition gives a refinement obligation, in which the composition, i.e., the collective of sub-systems, needs to conjointly fulfill the super system's contracts. Furthermore, the systems within a composition also need to be compatible with each other. Due to the over-approximating character of contracts, the Digital Twin tends to be imprecise, but sound for safety analyses. However, it is not possible to predict liveness properties. Note that proving the correctness of the system is sometimes infeasible, e.g., due to unavailability of the source code, complexity of the system (floating point, state explosion) or lack of knowledge of the environment. Additionally, we do not assume that the system is always compliant with the contract, especially, when the contracts are automatically mined from observation, or extracted from natural language and not carefully designed by the system creator and operators. This increases the need for a robust update that incrementally tighten the coupling between the instance of the system and Digital Twin built from contracts.

2.3. Update

There are several update operations. The most simple one is detecting the parameterization of the instance, i.e., determining the values of the different inputs.

Moreover, the assumptions and guarantees in the contracts at the top-level and sub-systems needs to be validated during operation to ensure that the system operates in its expected boundaries. In particular, we check whether a system is correctly invoked by its outer context or environment, and whether it behaves as specified. To achieve this, Runtime Verification, such as [6] for LTL, can be used. However, from Runtime Verification we can only learn about violation of or adherence to a property.

Normally, we want to establish the correctness of a system statically, but some properties are too hard to be established in advance. In the best case, unexpected specification-violating runtime traces are recorded to enable specification mining. Specification mining is the discipline to extract LTL properties from traces. For example, Tuxedo [7] is a pattern-based approach, that instantiates LTL patterns with state formulas and validates them against the traces. An instantiated pattern must be fulfilled by a specified ratio of the observed traces to be considered valid. The valid instantiated patterns help to adjust the contracts. For example, in the case of a contract violation, we might consider weakening the assumptions of a system. On the other hand, specification mining may also be applied where the contracts are fulfilled. In this case, it can lead to a tightening of the contracts for a specific configuration or environment. Besides of mining specifications from recorded traces, there are tools for extracting LTL properties from natural languages, e.g. requirement documentation or user commands [8].

2.4. Prediction

The prediction operation of our Digital Twin can be reduced to the well-established model-checking problem which allows us to verify the validity of a (LTL) property in a given Kripke structure. This covers the validation that each contract is adhered to by the associated system, and that systems are composed in a valid fashion.

But we benefit from the information learned during the Update operation: We can testify the contract adherence specifically in the used system configuration, and thus, we can save verification run-time without suffering a loss of verification validity for a specific instance. Furthermore, it may be beneficial to limit the verification process only to the recorded and mined environment of the instance to be validated.

Of course, the Digital Twin allows to conduct "What-If" analyses by altering the parameter of a system, or implementing it into a different environment. The latter simply requires changing the mined properties. This also requires tracking the origin of the formulas within the contracts.

But we are not limited to model-checking. Testing or simulation are also in reach by using the contracts. In particular, formal contracts enable testing and simulation even for software which is not executable in a simulation environment (e.g., due to inaccessible resources). There are also more sophisticated techniques. For example, *Program Repair* considers altering programs such that the program fulfills its contract. For this, the source code of the program is mutated until a suitable candidate is found [9, 10]. A similar discipline is *Parameter Synthesis* [11].

Limitations By using assume-guarantee contracts and LTL as the specification language, we focus on describing behavior on a certain abstraction level. This allows us to reason well on the possibility to reach bad system states, but other properties are not predictable. For example, security properties that are expressible as reachability are covered, i.e., integrity of the system, but confidentiality (e.g., secure information flow), and availability (liveness properties) are not. Moreover, runtime predictions, e.g., worst case execution time (WCET), are also not in our setup, due to the abstraction of time in LTL.

2.4.1. Other contract languages.

Note that OCRA uses LTL, but its approach is not limited to a particular contract grammar. There are many variants of temporal logics. For example, Metric Temporal Logic (MTL) extends LTL to include real-time capabilities [12]. In MTL, temporal operators have an additional time span in which the formula needs to be fulfilled. MTL formulas are evaluated on "data words"—event traces with an explicit time value. MTL is well studied for runtime verification [13, 14] and specification mining [15]. Additionally, Runtime Verification for MTL can also be quantitative by measuring how large the violation of the specification is [16].

Temporal logic formulas tend to be very hard to understand. Therefore, an additional goal is comprehensibility of the contracts and thus the Digital Twin. For example, Generalized Test Tables (GTTs) [17] are an engineer-friendly specification language derived from test tables used in the automation industry. A GTT describes a behavior in a particular scenario without the aspiration to be a complete specification. The rows in a GTT are the sequential steps of a test protocol, where each step consists of multiple assumptions and assertions (the table columns) for each input-output variable. Due to the table-based format, a fine-grained runtime monitoring is possible [18] in which the violation of single constraints can be tracked. Specification mining is currently not available, but might be adopted from approaches for learning finite-state machine based specifications (e.g., [19]).

3. Closing Remarks

In this paper, we present the idea of using formal contracts – existing in current design-by-contract approaches – for the representation of Digital Twins. Due to tool support, we focus on reactive systems and LTL, but the principle is also applicable to batch systems. For example, the Java Modeling Language (JML) [20] is an established specification language for Java source code with support of deductive verification [21] and runtime verification [22]. Specification mining (of method contracts) based on runtime information is currently not well-researched.

Software and its refinement. When we fade out the physical environment and concentrate on the software, we can state that the software, as it is digital, is itself the most precise definition of its behavior, and can also been seen as its own contract. Every other contract for a system over-approximates. But the abstraction of contracts is needed, as it allows us to hide the complexity of the implementation. Indeed, more abstract software models are often used in software engineering: For example, the buildability of a user-configurable product is defined by a feature model. Feature models reflect the compatibility of software modules, which also depends on the behavior of the software.

Evolution of Digital Twins with Relational Verification. For our framework, we have only considered the classical specification and verification of single traces. A possible extension is the verification of relational properties [23] (or multi-properties). A relational

property describes the relationship between multiple runs of the same or different systems. For example, regression verification – a relaxed program equivalence – is a relational property which helps to identify the introduction of unintended behavior during the co-evolution of hard- and software. Regression verification requires a description of the relationship between the common behavior of both systems [24]. As the regression contracts help with the co-evolution by coupling the old to the new version, they also support the evolution step of the Digital Twins by coupling mined knowledge from the old twin to the new twin.

Acknowledgments

This work was supported by funding of the Helmholtz Association (HGF) through the Competence Center for Applied Security Technology (KASTEL) and by the research project SofDCar (19S21002), which is funded by the German Federal Ministry for Economic Affairs and Climate Action

References

[1] A. Löcklin, M. Müller, T. Jung, N. Jazdi, D. White, M. Weyrich, Digital twin for verification and validation of industrial automation systems – a survey, in: 2020 25th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA), volume 1, 2020, pp. 851–858. doi:10.1109/ETFA46521.2020.9212051.

[2] G. D. Giacomo, M. Lenzerini, TBox and ABox reasoning in expressive description logics, in: L. Padgham, E. Franconi, M. Gehrke, D. L. McGuinness, P. F. Patel-Schneider (Eds.), Proceedings of the 1996 International Workshop on Description Logics, November 2-4, 1996, Cambridge, MA, USA, volume WS-96-05 of *AAAI Technical Report*, AAAI Press, 1996, pp. 37–48.

[3] N. Halbwachs, Synchronous programming of reactive systems, in: A. J. Hu, M. Y. Vardi (Eds.), Computer Aided Verification, 10th International Conference, CAV '98, Vancouver, BC, Canada, June 28 - July 2, 1998, Proceedings, volume 1427 of *Lecture Notes in Computer Science*, Springer, 1998, pp. 1–16. doi:10.1007/BFb0028726.

[4] M. Lohstroh, C. Menard, S. Bateni, E. A. Lee, Toward a Lingua Franca for deterministic concurrent systems, ACM Trans. Embed. Comput. Syst. 20 (2021). doi:10.1145/3448128.

[5] A. Cimatti, M. Dorigatti, S. Tonetta, OCRA: A tool for checking the refinement of temporal contracts, in: E. Denney, T. Bultan, A. Zeller (Eds.), 2013 28th IEEE/ACM International Conference on Automated Software Engineering, ASE 2013, Silicon Valley, CA, USA, November 11-15, 2013, IEEE, 2013, pp. 702–705. doi:10.1109/ASE.2013.6693137.

[6] A. Bauer, M. Leucker, C. Schallhart, Runtime verification for LTL and TLTL, ACM Trans. Softw. Eng. Methodol. 20 (2011). doi:10.1145/2000799.2000800.

[7] C. Lemieux, D. Park, I. Beschastnikh, General LTL specification mining (T), in: M. B. Cohen, L. Grunske, M. Whalen (Eds.), 30th IEEE/ACM International Conference

on Automated Software Engineering, ASE 2015, Lincoln, NE, USA, November 9-13, 2015, IEEE Computer Society, 2015, pp. 81–92. doi:10.1109/ASE.2015.71.

[8] C. Wang, C. Ross, Y. Kuo, B. Katz, A. Barbu, Learning a natural-language to LTL executable semantic parser for grounded robotics, in: J. Kober, F. Ramos, C. J. Tomlin (Eds.), 4th Conference on Robot Learning, CoRL 2020, 16-18 November 2020, Virtual Event / Cambridge, MA, USA, volume 155 of *Proceedings of Machine Learning Research*, PMLR, 2020, pp. 1706–1718.

[9] M. Brizzio, R. Degiovanni, M. Cordy, M. Papadakis, N. Aguirre, Automated repair of unrealisable LTL specifications guided by model counting, CoRR abs/2105.12595 (2021). arXiv:2105.12595.

[10] V. Mironovich, M. Buzdalov, V. Vyatkin, Automatic generation of function block applications using evolutionary algorithms: Initial explorations, in: 15th IEEE International Conference on Industrial Informatics, INDIN 2017, Emden, Germany, July 24-26, 2017, IEEE, 2017, pp. 700–705. doi:10.1109/INDIN.2017.8104858.

[11] P. Bezdek, N. Benes, J. Barnat, I. Cerná, LTL parameter synthesis of parametric timed automata, in: R. D. Nicola, E. Kühn (Eds.), Software Engineering and Formal Methods - 14th International Conference, SEFM 2016, Held as Part of STAF 2016, Vienna, Austria, July 4-8, 2016, Proceedings, volume 9763 of *Lecture Notes in Computer Science*, Springer, 2016, pp. 172–187. doi:10.1007/978-3-319-41591-8_12.

[12] R. Koymans, Specifying real-time properties with metric temporal logic, Real Time Syst. 2 (1990) 255–299. doi:10.1007/BF01995674.

[13] P. Thati, G. Rosu, Monitoring algorithms for metric temporal logic specifications, in: K. Havelund, G. Rosu (Eds.), Proceedings of the Fourth Workshop on Runtime Verification, RV@ETAPS 2004, Barcelona, Spain, April 3, 2004, volume 113 of *Electronic Notes in Theoretical Computer Science*, Elsevier, 2004, pp. 145–162. doi:10.1016/j.entcs.2004.01.029.

[14] H. Ho, J. Ouaknine, J. Worrell, Online monitoring of metric temporal logic, in: B. Bonakdarpour, S. A. Smolka (Eds.), Runtime Verification - 5th International Conference, RV 2014, Toronto, ON, Canada, September 22-25, 2014. Proceedings, volume 8734 of *Lecture Notes in Computer Science*, Springer, 2014, pp. 178–192. doi:10.1007/978-3-319-11164-3_15.

[15] B. Hoxha, A. Dokhanchi, G. Fainekos, Mining parametric temporal logic properties in model-based design for cyber-physical systems, Int. J. Softw. Tools Technol. Transf. 20 (2018) 79–93. doi:10.1007/s10009-017-0447-4.

[16] A. Dokhanchi, B. Hoxha, G. Fainekos, On-line monitoring for temporal logic robustness, in: B. Bonakdarpour, S. A. Smolka (Eds.), Runtime Verification - 5th International Conference, RV 2014, Toronto, ON, Canada, September 22-25, 2014. Proceedings, volume 8734 of *Lecture Notes in Computer Science*, Springer, 2014, pp. 231–246. doi:10.1007/978-3-319-11164-3_19.

[17] B. Beckert, M. Ulbrich, B. Vogel-Heuser, A. Weigl, Generalized test tables: A domain-specific specification language for automated production systems, in: H. Seidl, Z. Liu, C. S. Pasareanu (Eds.), Theoretical Aspects of Computing - ICTAC 2022 - 19th International Colloquium, Tbilisi, Georgia, September 27-29, 2022, Proceedings,

volume 13572 of *Lecture Notes in Computer Science*, Springer, 2022, pp. 7–13. doi:10.1007/978-3-031-17715-6_2.

[18] A. Weigl, M. Ulbrich, S. S. Tyszberowicz, J. Klamroth, Runtime verification of generalized test tables, in: A. Dutle, M. M. Moscato, L. Titolo, C. A. Muñoz, I. Perez (Eds.), NASA Formal Methods - 13th International Symposium, NFM 2021, Virtual Event, May 24-28, 2021, Proceedings, volume 12673 of *Lecture Notes in Computer Science*, Springer, 2021, pp. 358–374. doi:10.1007/978-3-030-76384-8_22.

[19] T. B. Le, D. Lo, Deep specification mining, in: F. Tip, E. Bodden (Eds.), Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2018, Amsterdam, The Netherlands, July 16-21, 2018, ACM, 2018, pp. 106–117. doi:10.1145/3213846.3213876.

[20] G. T. Leavens, Y. Cheon, C. Clifton, C. Ruby, D. R. Cok, How the design of JML accommodates both runtime assertion checking and formal verification, Sci. Comput. Program. 55 (2005) 185–208. doi:10.1016/j.scico.2004.05.015.

[21] W. Ahrendt, B. Beckert, R. Bubel, R. Hähnle, P. H. Schmitt, M. Ulbrich (Eds.), Deductive Software Verification - The KeY Book - From Theory to Practice, volume 10001 of *Lecture Notes in Computer Science*, Springer, 2016. doi:10.1007/978-3-319-49812-6.

[22] F. Hussain, G. T. Leavens, temporaljmlc: A JML runtime assertion checker extension for specification and checking of temporal properties, in: J. L. Fiadeiro, S. Gnesi, A. Maggiolo-Schettini (Eds.), 8th IEEE International Conference on Software Engineering and Formal Methods, SEFM 2010, Pisa, Italy, 13-18 September 2010, IEEE Computer Society, 2010, pp. 63–72. doi:10.1109/SEFM.2010.15.

[23] B. Beckert, M. Ulbrich, Trends in relational program verification, in: P. Müller, I. Schaefer (Eds.), Principled Software Development - Essays Dedicated to Arnd Poetzsch-Heffter on the Occasion of his 60th Birthday, Springer, 2018, pp. 41–58. doi:10.1007/978-3-319-98047-8_3.

[24] A. Weigl, M. Ulbrich, S. Cha, B. Beckert, B. Vogel-Heuser, Relational test tables: A practical specification language for evolution and security, in: K. Bae, D. Bianculli, S. Gnesi, N. Plat (Eds.), FormaliSE@ICSE 2020: 8th International Conference on Formal Methods in Software Engineering, Seoul, Republic of Korea, July 13, 2020, ACM, 2020, pp. 77–86. doi:10.1145/3372020.3391566.