

Leveraging Runtime Verification for the Monitoring of Digital Twins

Sylvain Hallé¹, Chukri Soueidi² and Yliès Falcone²

¹Laboratoire d'informatique formelle, Université du Québec à Chicoutimi, Canada

²Laboratoire d'informatique de Grenoble, Université Grenoble Alpes, France

Abstract

The paper considers the problem of discovering divergences between the actions of a digital twin and those of its real-world counterpart. It observes the similarities between this problem and an existing field of formal methods called Runtime Verification (RV), and suggests leveraging and adapting RV techniques to this effect. Concretely, three important aspects of the problem are identified and for which both theoretical and practical challenges must be addressed.

Keywords

digital twins, runtime verification, stream processing

1. Introduction

A *digital twin* is a virtual representation of a real-world entity [1]; it is often presented as a predictive instrument, by enabling one to simulate multiple possible outcomes of a real-world entity. In such a context, it is essential that the digital twin exhibits behavior that is faithful to that of the system it seeks to mimic. Any significant and sustained discrepancy between the twin and its concrete counterpart can lead to incorrect predictions, false diagnoses, and generally to an incorrect perception of the operation of the real system. Differences between the operation of a twin and the real-world entity must therefore be monitored and addressed in real time as they occur.

The process of detecting deviations can be summarized as illustrated in Figure 1. A real-world entity E is given inputs I , which can consist of controllable (i.e. user-defined) values, as well as uncontrollable (i.e. environmental) objects, in addition to any reading related to the entity's current internal state. The entity reacts to these inputs by producing outputs O_E ; again, these outputs can be actions directly performed by the entity, or measured values of the entity's state or the environment. In parallel, the inputs are recorded and fed to a digital twin T , which simulates the real-world entity and produces its own outputs O_T . The observed output and the synthetic output are then fed to a comparison procedure C , which decides whether they are in agreement (\top) or not (\perp).


In this paper, we reveal the similarities that this problem shares with a research topic crossing the fields of software engineering and formal methods, called *runtime verification* (RV) [2]. Over

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

✉ shalle@acm.org (S. Hallé); chukri.a.soueidi@inria.fr (C. Soueidi); ylies.falcone@inria.fr (Y. Falcone)



© 2023 Copyright for this paper by its authors. Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0).

 CEUR Workshop Proceedings (CEUR-WS.org)

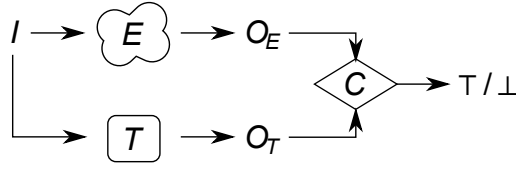


Figure 1: Comparing the behavior of a real-world entity with that of a digital twin.

the years, RV has been successfully applied to various use cases, ranging from the monitoring of aerial drones [3] to the detection of bugs in video games [4]. We identify elements of the problem that require adaptations in order to leverage RV for digital twin monitoring, and suggest possible ways in which these techniques could be used in this context.

2. A Property-Based Approach

Runtime verification is the discipline of computer science where an object called a *monitor* is used to observe the behavior of another program. At various moments, instrumented codepoints relay information about the program’s actions and state to the monitor, producing a sequence of data elements called “events”, denoted by $\bar{\sigma} = \sigma_1, \sigma_2, \dots$. The monitor compares this event trace against a formal specification φ of what constitutes a correct execution; any violation of the specification is reported on-the-fly, as the monitored program executes. Research on runtime verification has produced a variety of monitors supporting a large class of specification languages [5, 6, 7, 8, 9].

A Passive Operation As in RV, the monitoring of digital twins is mostly a *passive* procedure: one is not concerned with generating inputs that drive the system, as is the case for conformance testing [10, 11]; this is typically done by calculating the sequences of inputs that have the potential to reveal a violation of the finite-state machine specification by the system. However, in the context of digital twins, one rarely has the possibility of sending unlimited sequences of inputs to the concrete entity to ensure the conformity of the model, due to their associated cost (in terms of time and resources). What is more, some scenarios may involve compliance checking in situations where the actual system is damaged, and may be excluded from the test cases from the outset for this reason. Realistically, the best that can be done is to observe the behavior of the actual system in its normal operation, and to make the most of these observations to detect any discrepancies as they occur.

Properties on Digital Twins A first possible application of runtime verification consists of an indirect comparison between the twin’s behavior and that of the real-world entity. It supposes the existence of conditions $\varphi_1, \dots, \varphi_n$, which are known to be true for all executions of the digital twin; these properties, acting as a form of “guarantee,” are extracted from the twin beforehand by an arbitrary procedure G , as illustrated in Figure 2. These properties, in turn, can be converted into monitors that observe the operation of the real-world entity in real time. Any observed sequence that violates one of the φ_i is, by definition, a sequence that cannot be

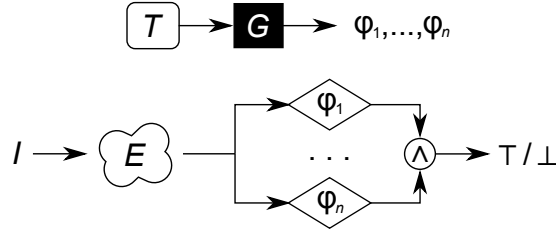


Figure 2: Using properties on input/output sequences to detect divergences in the execution of a twin.

produced by the digital twin for the same inputs, and thus indicates a divergence between the twin and its real-world counterpart.

Note that in such a scenario, the twin itself does not need to be “executed” —that is, it is unnecessary to have the twin generate one or more traces for comparison with the real-world entity’s output. This approach can present advantages in cases where running a twin may incur a high cost. Instead of a constant synchronization between the twin and the real-world entity, the lighter monitors may observe the output of the entity as long as it corresponds to expected behavior. Further analysis (and possible adaptation of the twin) is only required when one of the properties is violated by the observations. This, in itself, may require substantial analysis to determine which are the relevant φ_i that need to be monitored.

Declarative Definition of a Twin In this first suggested mode of operation, the detection of divergences is *sound*: a violation of one of the properties indicates a genuine discrepancy between the twin and the real-world entity. The detection is *complete* if any sequence satisfying $\bigwedge \varphi_i$ is also a possible (valid) sequence produced by the twin. In such a case, the twin’s behavior is completely captured by the conditions φ_i that serve as definitions. Thus, one can imagine specifying the operation of the twin in a *declarative* way, as opposed to a “procedural” or “imperative” way. Instead of programming the twin as an executable object that receives inputs and produces outputs, the execution of the twin is driven by a satisfiability (SAT) solver: given a sequence of inputs, the solver finds a sequence of outputs satisfying the properties, and returns that sequence as the twin’s reaction to the inputs. Such a declarative approach has already been attempted to simulate the execution of web services from temporal logic specifications [12].

3. Generalizing Runtime Verification

The previous approach is the most direct application of runtime verification to digital twins. However, its soundness rests on the hypothesis that a set of properties of the twin can be extracted and used as formal guarantees on its execution. For a twin that is defined procedurally, those properties can be deduced from its implementation (for example by defining properties manually and verifying them through model checking [13], or by observing multiple executions of the twin and deducing a formal model of its execution using process mining [14]).

However, obtaining these guarantees may be a complex process, and completely capturing the behavior of the twin in such a way may not be a realistic assumption. Another possibility

consists of handling the twin as a black box, and to directly compare its output to that of the real-world entity (for a given input sequence), as described in Figure 1. A second point that this article puts forward is that this comparison can be framed as a generalization of runtime verification.

Monitoring Two Traces Contrary to RV, monitoring twins involves not one, but (at least) two traces at the same time (O_E and O_T). The “property” that needs to be evaluated in such a case correlates the events observed in both traces. This is a particular case of what is called a *hyperproperty* [15, 16]; while a property determines whether a trace is valid or not in isolation, in hyperproperties traces are valid or not based on their relation with other traces. It shall be noted that this operation, taken in its most abstract form, can be any calculation; as we shall discuss below, it is not restricted to the pairwise comparison of events at matching indices in both traces, and can involve arbitrary constraints on the values and ordering of events at various locations.

Expressiveness As a matter of fact, an anticipated challenge for the leveraging of RV techniques to digital twin monitoring is the relatively low expressiveness of the notations they use as their specification language. A recent taxonomy of existing RV approaches highlights the fact that many of them use formalisms based on propositional temporal logic or finite-state automata [17]. Some of them have support for quantification over data values, or implement basic forms of aggregation such as sum or average [18, 5, 19, 9]. However, these languages are ill-suited in their present form to handle the rich event types and complex (and often numerical) relationships that can involve the values they contain –and most importantly, specification languages for hyperproperties are even more restricted than for classical properties.

Towards Stream Processing In this context, it might be desirable to expand the vision of the problem and to consider it as a more general form of complex event processing (CEP) [20, 21]. CEP is typically concerned with data-rich events, and considers arbitrary calculations over these events in order to produce higher-level (i.e. “complex”) events. Its more general computational model could be harnessed in order to express the potentially intricate operations required for uncovering discrepancies between the output of a digital twin and that of its real-world counterpart. Previous works have shown how CEP engines can evaluate properties specified in many languages used in RV and extend them with additional constructs [22], making them good candidates to address the expressiveness issues mentioned above.

4. Qualifying Divergence

Another important challenge lies in the fact that not all divergences between an entity and its twin are meaningful and indicative of a problem. There are situations where discrepancies between an entity and its twin are expected, if not unavoidable, and still do not represent any malfunction, modeling error, or significant drift between a twin and its real-world counterpart.

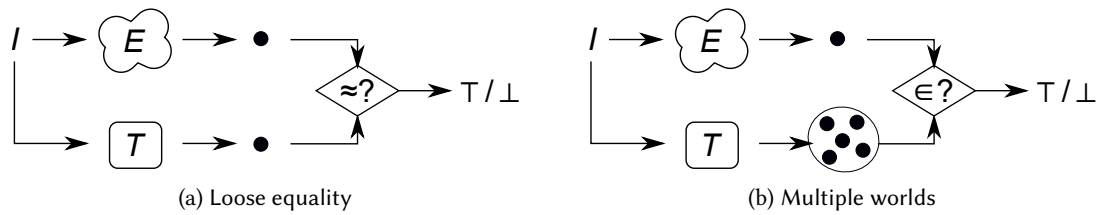


Figure 3: Two possible ways of comparing the output of a twin with that of a real-world system while tolerating some divergence.

Sources of Divergence A first such case is caused by *uncertainty* in measurements. A physical entity will typically have its environment and behavior measured by sensors, whose output is intrinsically tainted with uncertainty, bias, or even spurious drop-outs. Thus one cannot expect a strict equality between values produced (or predicted) by a twin and those measured in a real-world system. It turns out that several works in the field of RV have addressed the issue of verifying properties in the presence of missing or imprecise events [23, 24, 25], which could be leveraged to the context of digital twins.

A second source of divergence may be caused by different *interleavings* in the events produced by a twin and its counterpart. This can happen when events produced by multiple components of the system happen more or less simultaneously, and are arbitrarily flattened to a particular ordering which may differ depending on uncontrollable or external factors. Instrumentation is sometimes designed on purpose to lose some ordering information, as is the case in some cyber-physical systems [26].

A final source of divergences comes from *under-specification*. In some cases, it is expected that the digital twin is run from a coarse-grained description of the real-world entity that does not totally capture its internal state. This typically shows up as the system appearing to operate non-deterministically, producing different outputs in what are apparently identical input conditions. In some other cases, non-determinism may simply arise from the fact that multiple possible (and equally acceptable) outcomes are possible for the same set of initial conditions.

For all these reasons, it is unrealistic that the comparison procedure C shown in Figure 1 looks at O_E and O_T and simply expects both to be identical. We identify two ways of dealing with this issue, one being the opposite of the other, and illustrated in Figure 3.

Single World, Loose Equality A first possibility, illustrated in Figure 3a is to let a digital twin produce a unique output for a given input (i.e. a “single possible world”). However, it is allowed (and even expected) that this output O_T differs slightly from O_E ; therefore, the comparison procedure C does not look for strict equality between the two streams, but rather evaluates a relaxed property. For example, for a system producing a stream of numerical values, one may expect that the running average over a sliding window of k values is the same, but not the individual events produced by both systems. In mathematical terms, the comparison criterion is an equivalence relation that is looser than equality. For numerical values, this can be likened to fuzzy comparators [27].

This mode of operation brings challenges of its own. First, an appropriate equivalence relation must be defined, and it is expected that such relation be specific to each problem domain. Second is the necessity of evaluating this relation efficiently at runtime, over two streams of events that are progressively produced by both the twin and the real-world entity. Deviations should be reported on-the-fly, as obviously one cannot wait for the executions to complete before starting the comparison. This seemingly innocuous observation makes it difficult to use well-known string distance criteria, such as Levenshtein distance [28], which have a high computational complexity and typically expect strings to be completely known in advance to calculate their value.

Multiple Worlds, Strict Equality An alternate approach, illustrated in Figure 3b, is to allow a twin to produce multiple possible outputs for a given input. Each of these outputs corresponds to one of the “possible worlds” the system can be in for given input conditions. Strict equality is then sought between the real-world entity’s output, and one of those possible worlds¹. In this setting, the multiple possible worlds can either be represented explicitly (as an enumeration of all possible outputs) or implicitly (through an abstract property that it satisfied by all possible worlds). For example, a numerical value associated with a precision interval counts as an abstract representation of multiple exact numerical values.

The multi-world trace semantics has been used to address the question of runtime verification with uncertainty [23, 29], and could be adapted to the problem of stream comparison for digital twins.

5. Conclusion

In this paper, we have highlighted the connections that can be made between the question of detecting discrepancies between a digital twin and a concrete entity, and the problem of runtime verification already studied in the community of software engineering and formal methods. Although they present clear similarities, these two problems are nevertheless distinct, and some adaptation is therefore necessary in order to leverage existing runtime verification techniques to the particular context of digital twins.

The article identified several research directions aimed at enabling real-time divergence checking using RV techniques, which will be explored in more detail in future work. Among these, we note the design of specification languages that are more expressive and appropriate to the problem of digital twins, as well as the definition of trace comparison metrics that are less strict than a simple position-by-position equality.

References

- [1] M. Grieves, J. Vickers, Digital Twin: Mitigating Unpredictable, Undesirable Emergent Behavior in Complex Systems, Springer, 2017, pp. 85–113.

¹Stated otherwise, the entity’s output must be included in the possible outputs produced by the twin.

- [2] E. Bartocci, Y. Falcone, A. Francalanza, G. Reger, Introduction to runtime verification, in: E. Bartocci, Y. Falcone (Eds.), *Lectures on Runtime Verification - Introductory and Advanced Topics*, volume 10457 of *Lecture Notes in Computer Science*, Springer, 2018, pp. 1–33.
- [3] P. Moosbrugger, K. Y. Rozier, J. Schumann, R2U2: monitoring and diagnosis of security threats for unmanned aerial systems, *Formal Methods Syst. Des.* 51 (2017) 31–61.
- [4] S. Varvaressos, K. Lavoie, S. Gaboury, S. Hallé, Automated bug finding in video games: A case study for runtime monitoring, *Comput. Entertain.* 15 (2017) 1:1–1:28.
- [5] B. D’Angelo, S. Sankaranarayanan, C. Sánchez, W. Robinson, B. Finkbeiner, H. B. Sipma, S. Mehrotra, Z. Manna, LOLA: runtime monitoring of synchronous systems, in: *TIME*, IEEE Computer Society, 2005, pp. 166–174.
- [6] P. O. Meredith, D. Jin, D. Griffith, F. Chen, G. Rosu, An overview of the MOP runtime verification framework, *Int. J. Softw. Tools Technol. Transf.* 14 (2012) 249–289.
- [7] C. Colombo, G. J. Pace, G. Schneider, LARVA – safer monitoring of real-time Java programs (tool paper), in: D. V. Hung, P. Krishnan (Eds.), *SEFM*, IEEE Computer Society, 2009, pp. 33–37.
- [8] G. Reger, H. C. Cruz, D. E. Rydeheard, Marq: Monitoring at runtime with QEA, in: C. Baier, C. Tinelli (Eds.), *TACAS*, volume 9035 of *Lecture Notes in Computer Science*, Springer, 2015, pp. 596–610.
- [9] D. A. Basin, F. Klaedtke, S. Marinovic, E. Zalinescu, Monitoring of temporal first-order properties with aggregations, *Formal Methods Syst. Des.* 46 (2015) 262–285.
- [10] D. Lee, M. Yannakakis, Principles and methods of testing FSMs: A survey, *Proceedings of the IEEE* 84 (1996) 1089–1123.
- [11] C. Constant, T. Jéron, H. Marchand, V. Rusu, Integrating formal verification and conformance testing for reactive systems, *IEEE Trans. Software Eng.* 33 (2007) 558–574.
- [12] S. Hallé, Model-based simulation of SOAP web services from temporal logic specifications, in: I. Perseil, K. K. Breitman, R. Sterritt (Eds.), *ICECCS*, IEEE Computer Society, 2011, pp. 95–104.
- [13] C. Baier, J.-P. Katoen, *Principles of Model Checking*, MIT Press, 2008.
- [14] W. van der Aalst, *Process Mining: Data Science in Action*, Springer, 2016.
- [15] M. R. Clarkson, F. B. Schneider, Hyperproperties, in: *CSF*, IEEE Computer Society, 2008, pp. 51–65.
- [16] B. Finkbeiner, C. Hahn, M. Stenger, L. Tentrup, Monitoring hyperproperties, *Formal Methods Syst. Des.* 54 (2019) 336–363.
- [17] Y. Falcone, S. Krstic, G. Reger, D. Traytel, A taxonomy for classifying runtime verification tools, *Int. J. Softw. Tools Technol. Transf.* 23 (2021) 255–284.
- [18] S. Hallé, R. Villemaire, Runtime enforcement of web service message contracts with data, *IEEE Trans. Serv. Comput.* 5 (2012) 192–206.
- [19] L. Convent, S. Hungerecker, M. Leucker, T. Scheffel, M. Schmitz, D. Thoma, TeSSLa: Temporal stream-based specification language, in: T. Massoni, M. R. Mousavi (Eds.), *SBMF*, volume 11254 of *Lecture Notes in Computer Science*, Springer, 2018, pp. 144–162.
- [20] S. Hallé, *Event Stream Processing With BeepBeep 3: Log Crunching and Analysis Made Easy*, Presses de l’Université du Québec, 2018.
- [21] Event stream processing (ESP), in: L. Liu, M. T. Özsu (Eds.), *Encyclopedia of Database*

- Systems, Springer US, 2009, p. 1064.
- [22] S. Hallé, R. Houry, Writing domain-specific languages for beepbeep, in: C. Colombo, M. Leucker (Eds.), Runtime Verification - 18th International Conference, RV 2018, Limassol, Cyprus, November 10-13, 2018, Proceedings, volume 11237 of *Lecture Notes in Computer Science*, Springer, 2018, pp. 447–457. doi:10.1007/978-3-030-03769-7_27.
 - [23] R. Taleb, R. Houry, S. Hallé, Runtime verification under access restrictions, in: S. Bludze, S. Gnesi, N. Plat, L. Semini (Eds.), FormaliSE@ICSE, IEEE, 2021, pp. 31–41.
 - [24] Y. Joshi, G. M. Tchamgoue, S. Fischmeister, Runtime verification of LTL on lossy traces, in: A. Seffah, B. Penzenstadler, C. Alves, X. Peng (Eds.), SAC, ACM, 2017, pp. 1379–1386.
 - [25] D. A. Basin, F. Klaedtke, S. Marinovic, E. Zalinescu, Monitoring compliance policies over incomplete and disagreeing logs, in: S. Qadeer, S. Tasiran (Eds.), RV, volume 7687 of *Lecture Notes in Computer Science*, Springer, 2012, pp. 151–167.
 - [26] S. Wang, A. Ayoub, O. Sokolsky, I. Lee, Runtime verification of traces under recording uncertainty, in: S. Khurshid, K. Sen (Eds.), RV, volume 7186 of *Lecture Notes in Computer Science*, Springer, 2011, pp. 442–456.
 - [27] A. Kaufmann, M. M. Gupta, Introduction to Fuzzy Arithmetic: Theory and Applications, Van Nostrand Reinhold, 1991.
 - [28] G. Navarro, A guided tour to approximate string matching, ACM Comput. Surv. 33 (2001) 31–88.
 - [29] M. Leucker, C. Sánchez, T. Scheffel, M. Schmitz, D. Thoma, Runtime verification for timed event streams with partial information, in: B. Finkbeiner, L. Mariani (Eds.), RV, volume 11757 of *Lecture Notes in Computer Science*, Springer, 2019, pp. 273–291.