Pushing runtime verification to the limit: May process semantics be with us*

Dario Della Monica[1] and Adrian Francalanza[2]

[1]University of Udine, Italy
[2]University of Malta, Malta
[1]`dario.dellamonica@uniud.it`
[2]`adrian.francalanza@um.edu.mt`

Abstract

We propose a combined approach that permits automated formal verification to be spread across the pre- and post-deployment phases of a system development, with the aim of calibrating the management of the verification burden. Our approach combines standard *model checking* methods with *runtime verification*, a relatively novel formal technique that verifies a system during its execution. We carry out our study in terms of the Hennessy-Milner Logic, a branching-time logic for specifying reactive system correctness. Whereas we will be mainly concerned with limiting the model checking verification burden, runtime verification has been shown to handle a strict subset of the expressible properties in our logic of study, posing constraints on what can be shifted to the post-deployment phase. We present a solution, based on modal transition systems and modal refinement, for the fragment of the Hennessy-Milner Logic devoid of recursion, i.e., without least and greatest fixpoint operators.

Introduction *Model checking* (MC) [17] is a widely accepted *pre-deployment* verification technique that checks whether a system satisfies or violates a property by potentially analysing *all* the possible system behaviours. By contrast, *runtime verification* (RV) [30, 11] is a lightweight verification technique aimed at mitigating scalability issues, such as the state explosion problem, typically associated with traditional verification techniques like MC. RV attempts to infer the satisfaction (or violation) of a correctness property from the analysis of the *current execution* of the system under scrutiny using monitors [22, 23]. It is thus performed *post-deployment* (on actual system execution), which is appealing for component-based applications (parts of which may not be available for analysis pre-deployment), as well as for dynamic settings such as mobile computing (where components are downloaded and installed at runtime). The technique has fostered a number of verification tools, *e.g.,* [9, 10, 18, 20, 27, 29, 32, 7, 8, 34] to name but a few, and has proved effective in various real-world scenarios [14, 37, 21].

Despite its advantages, RV is limited when compared to verification techniques such as MC because certain correctness properties cannot be verified at runtime [31, 33, 16, 26, 2, 3, 4]. For instance, MC

---

[*] This work is partially supported by the Italian INdAM-GNCS project *Metodi formali per tecniche di verifica combinata*, the Icelandic Research Fund project *TheoFoMon: Theoretical Foundations for Monitorability* (No.163406-051), and the project *BehAPI*, funded by the EU H2020 RISE programme under the Marie Skłodowska-Curie grant agreement (No.778233).

makes it possible to check for both *safety* and *liveness* properties, by providing either a positive or a negative answer, according to whether the system conforms with the specifications; RV, on the other hand, can only return a positive verdict for certain liveness properties (called co-safety properties [15, 4]) or a negative one for safety conditions. Moreover, RV induces a runtime overhead over the execution of a monitored system, which should ideally be kept to a minimum [30, 11].

Syntax

$$\begin{array}{rlllll}
\varphi,\phi \in \mu\text{HML} ::= & \mathsf{tt} & \text{(truth)} & \mid\ \mathsf{ff} & \text{(falsehood)} \\
\mid & \varphi\vee\phi & \text{(disjunction)} & \mid\ \varphi\wedge\phi & \text{(conjunction)} \\
\mid & \langle\alpha\rangle\varphi & \text{(possibility)} & \mid\ [\alpha]\varphi & \text{(necessity)} \\
\mid & \mathsf{min}\,X.\varphi & \text{(min. fixpoint)} & \mid\ \mathsf{max}\,X.\varphi & \text{(max. fixpoint)} \\
\mid & X & \text{(rec. variable)} & &
\end{array}$$

Semantics

$$\begin{array}{rlrl}
[\![\mathsf{tt},\rho]\!] & \overset{\mathbf{def}}{=} \textsc{Sta} & [\![\mathsf{ff},\rho]\!] & \overset{\mathbf{def}}{=} \emptyset \\
[\![\varphi_1{\vee}\varphi_2,\rho]\!] & \overset{\mathbf{def}}{=} [\![\varphi_1,\rho]\!]\cup[\![\varphi_2,\rho]\!] & [\![\varphi_1{\wedge}\varphi_2,\rho]\!] & \overset{\mathbf{def}}{=} [\![\varphi_1,\rho]\!]\cap[\![\varphi_2,\rho]\!] \\
[\![\langle\alpha\rangle\varphi,\rho]\!] & \overset{\mathbf{def}}{=} \left\{ s \mid \exists r.s \xrightarrow{\alpha} r \text{ and } r\in[\![\varphi,\rho]\!] \right\} & [\![[\alpha]\varphi,\rho]\!] & \overset{\mathbf{def}}{=} \left\{ s \mid \forall r.s \xrightarrow{\alpha} r \text{ implies } r\in[\![\varphi,\rho]\!] \right\} \\
[\![\mathsf{min}X.\varphi,\rho]\!] & \overset{\mathbf{def}}{=} \bigcap\{S\in\textsc{Sta} \mid [\![\varphi,\rho[X\mapsto S]]\!]\subseteq S\} & [\![\mathsf{max}X.\varphi,\rho]\!] & \overset{\mathbf{def}}{=} \bigcup\{S\in\textsc{Sta} \mid S\subseteq[\![\varphi,\rho[X\mapsto S]]\!]\} \\
[\![X,\rho]\!] & \overset{\mathbf{def}}{=} \rho(X) & &
\end{array}$$

Monitorable Fragments

$$\begin{array}{rllllll}
\theta,\vartheta \in \text{sHML} ::= & \mathsf{tt} & \mid\ \mathsf{ff} & \mid\ [\alpha]\theta & \mid\ \theta{\wedge}\vartheta & \mid\ \mathsf{max}\,X.\theta & \mid\ X \\
\pi,\varpi \in \text{cHML} ::= & \mathsf{tt} & \mid\ \mathsf{ff} & \mid\ \langle\alpha\rangle\pi & \mid\ \pi{\vee}\varpi & \mid\ \mathsf{min}\,X.\pi & \mid\ X
\end{array}$$

Figure 1: μHML Syntax and Semantics

Hennessy-Milner Logic with Recursion (μHML) RV's limits in terms of verifiable properties is evidenced more for branching-time logics, that are able to express properties describing behaviour over multiple system executions. In recent work [25, 26, 1, 3], one such branching-time logic called μHML [13, 5] is studied from an RV perspective. Figure 1 outlines the syntax of the logic μHML, along with its semantics, defined over a *Labelled Transition System* (LTS), *i.e.,* triples $\langle\textsc{Sta},\textsc{Act},\longrightarrow\rangle$ consisting of a set of *states* $s,r\in\textsc{Sta}$, a set of *actions* $\alpha\in\textsc{Act}$, and a *transition relation* between states labelled by actions, $s\xrightarrow{\alpha}r$; as in [5, 26], the semantic definition employs an *environment* from μHML logical variables, $\textsc{Vars}$, to sets of states, $\rho\in(\textsc{Vars}\rightharpoonup\mathcal{P}(\textsc{Sta}))$ (see Figure 1). One of the main contributions of [25] is the identification of an *expressively maximal, runtime-verifiable subset* of the logic, reported in Figure 1 as the grammar for sHML and cHML (see also [26, 24]); the authors show how these classes provide an easy syntactic check for determining whether a property satisfaction (or violation) can be determined using the RV technique. The cHML (resp., sHML) fragment of μHML is said to be *positively monitorable* (resp., *negatively monitorable*) [26, 4].

Extending the applicability of monitoring towards a combined verification We build on the findings of [25], with the aim of extending the applicability of RV to a larger class of μHML properties other than sHML $\cup$ cHML from Figure 1. Specifically, we propose a *combined approach* that permits automated formal verification to be spread across the pre- and post-deployment phases of a system development, with the aim of calibrating the management of the verification burden while combining the strengths of MC with those of RV. As an illustrative example, consider the μHML property (1) below, describing systems that *can* perform action a, prefix $\langle a\rangle(\ldots)$, and reach a state from where they *can either* perform action b, subformula $\langle b\rangle\mathsf{tt}$, *or* else *can never* perform action c, subformula $[c]\mathsf{ff}$.

$$\langle a\rangle(\langle b\rangle\mathsf{tt}\vee[c]\mathsf{ff}) \tag{1}$$

According to Figure 1, (1) turns out *not* to be runtime-verifiable because of the subformula $[c]\mathsf{ff}$; intuitively, whereas a system execution exhibiting action a followed by action b suffices to prove that the system satisfies (1), an RV monitor cannot determine whether a system can never produce action c after performing action a from the observation of only a *single* system execution [25]. However, property (1) can be expressed as the (logically equivalent) formula

$$(\langle a\rangle\langle b\rangle\mathsf{tt}) \vee (\langle a\rangle[c]\mathsf{ff}) \tag{2}$$

whereby we note that the subformula $\langle a\rangle\langle b\rangle\mathsf{tt}$ *is runtime verifiable*, according to [25, 26, 3]. We argue that reformulations such as (2) allow for a combined approach to verification, where part of the property, *e.g.,* the (smaller) subformula $\langle a\rangle[c]\mathsf{ff}$, can be checked prior system deployment using MC, and the remaining part of the property, *e.g.,* $\langle a\rangle\langle b\rangle\mathsf{tt}$, can be runtime-verified during system execution.

We therefore aim to devise *general* analysis techniques that reformulate any μHML formula into either conjunctions or disjunctions, *i.e.,* $\varphi_{\mathrm{RV}} \wedge \varphi_{\mathrm{MC}}$ or $\varphi_{\mathrm{RV}} \vee \varphi_{\mathrm{MC}}$, where φ_{RV} and φ_{MC} denote the runtime-verifiable and model-checkable formula components, respectively. From a software engineering perspective, we envisage at least two ways how this decomposition between pre- and post-deployment verification can be fruitful:

1. The ensuing combined approach may be used as a means to *minimise* the verification effort required *prior to the deployment* of a system. *E.g.,* in the case of (2), the model-checked subformula $\varphi_{\mathrm{MC}} = \langle a\rangle[c]\mathsf{ff}$ is *smaller* than the full formula (1), since we would be offloading a degree of verification onto the runtime phase when runtime-verifying for $\varphi_{\mathrm{RV}} = \langle a\rangle\langle b\rangle\mathsf{tt}$. Moreover, for disjunction decompositions such as (2), the satisfaction of φ_{MC} prior to deployment obviates the need for any runtime analysis, minimising runtime overheads (a dual argument applies for conjunction decompositions and φ_{MC} violations).

2. In settings where software correctness is desirable but not essential, a combined approach can be used as a means to circumvent full-blown MC. Specifically, instead of model-checking for (1), a system may be runtime-verified for $\varphi_{\mathrm{RV}} = \langle a\rangle\langle b\rangle\mathsf{tt}$ during its pilot launch, acting as a *vetting phase*: if φ_{RV} is satisfied during RV, this means that, by (2), (1) is satisfied as well; if not, we then proceed to model-check the system offline *wrt.* $\varphi_{\mathrm{MC}} = \langle a\rangle[c]\mathsf{ff}$.

A partial solution based on modal transition systems and modal refinement From a technical point of view, the problem amounts, to computing the maximal monitorable semantic fragment of a given μHML formula φ, that is, the formula ψ such that

- $\psi \in$ cHML (ψ is positively monitorable),
- $[\![\psi]\!] \subseteq [\![\varphi]\!]$ (every process that satisfies ψ also satisfies φ, i.e, ψ is a semantic fragment of φ),
- for every $\psi' \in$ cHML, we have that $[\![\psi']\!] \subseteq [\![\varphi]\!]$ implies $[\![\psi']\!] \subseteq [\![\psi]\!]$ (ψ is maximal)

Dually, for the case of negatively monitorable formulas (fragment sHML), we are interested in the minimal monitorable formula of which φ is a semantic fragment; we focus on the former formulation only since the latter one can be solved by exploiting the duality between the two fragments (as in [2]).

Instead of trying to obtain the desired formula through syntactic transformations, we adopt a semantic approach that proved itself successful for the logic HML, *i.e.,* the fragment of μHML devoid of fixpoint operators. We first transform the input formula into a *modal transition system* (*MTS*) [12], which can be thought of as graphical representations of formulas: MTSs suit our purpose particularly well because they are amenable to manipulations while preserving the information about the meaning of the formula. By using this technique, we can inherit known results from concurrency theory, such as the characterization and classification of several process semantics [19, 35, 36]. In particular, back and forth translations between HML formulas and MTSs are known [12], and a preorder over MTSs, called *modal refinement*, has been defined that captures the semantic relationship between HML specifications: an MTS M_1 precedes another MTS M_2 in this modal refinement preorder whenever the set of processes that satisfy φ_{M_1} (the translation of M_1 into an HML formula) contains set of processes that satisfy φ_{M_2}. Using such results we are able to identify the class of MTSs corresponding to the monitorable fragment of HML and to single out the MTS corresponding to the maximal monitorable semantic fragment of a given HML formula. Then, by employing the translation from MTSs back to HML formulas, we obtain the monitorable specification that we are looking for.

Future directions Extending our approach to the full μHML remains an open issue: for this purpose, MTSs should be extended with cycles so as to enable them to "mimic" fixpoint operators, which somehow correspond to recursion. Once this obstacle is solved, we can also investigate the application of our methods to the linear-time setting, where there are still formulas that are not monitorable [3, 4].

Another direction we intend to pursue is that of extending our techniques to settings with enriched monitoring capabilities. A number of these settings have recently been investigated for the logic μHML in the work [2] by considering monitoring setups with the ability to recognize when a process terminates, or the ability to infer the possible (1-step) actions from a specific state (even though the computation will then continue executing along only one of these actions). Using some of the aforementioned results from the field of concurrency theory and process semantics, our approach should extend in a straightforward manner to cope with the enhanced monitoring capabilities. More importantly, this study marries well with our aims for a multi-pronged verification methodology along the lines advocated in [6, 21, 28].

References

[1] L. Aceto, A. Achilleos, A. Francalanza, and A. Ingólfsdóttir. Monitoring for silent actions. In S. Lokam and R. Ramanujam, editors, *FSTTCS*, volume 93 of *LIPIcs*, pages 7:1–7:14, Dagstuhl, Germany, 2017. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[2] L. Aceto, A. Achilleos, A. Francalanza, and A. Ingólfsdóttir. A Framework for Parameterized Monitorability. In C. Baier and U. D. Lago, editors, *Proc. of the 21st International Conference on Foundations of Software Science and Computation Structures (FOSSACS)*, volume 10803 of *LNCS*, pages 203–220. Springer, 2018.

[3] L. Aceto, A. Achilleos, A. Francalanza, A. Ingólfsdóttir, and K. Lehtinen. Adventures in monitorability: From branching to linear time and back again. *Proceedings of the ACM on Programming Languages*, 3(POPL):52:1–52:29, 2019.

[4] L. Aceto, A. Achilleos, A. Francalanza, A. Ingólfsdóttir, and K. Lehtinen. An operational guide to monitorability. In *Software Engineering and Formal Methods - 17th International Conference, SEFM 2019, Oslo, Norway, September 18-20, 2019, Proceedings*, volume 11724 of *LNCS*, pages 433–453. Springer, 2019.

[5] L. Aceto, A. Ingólfsdóttir, K. G. Larsen, and J. Srba. *Reactive Systems: Modelling, Specification and Verification.* Cambridge Univ. Press, 2007.

[6] C. Artho, H. Barringer, A. Goldberg, K. Havelund, S. Khurshid, M. Lowry, C. Pasareanu, G. Roşu, K. Sen, W. Visser, and R. Washington. Combining test case generation and runtime verification. *Theoretical Computer Science*, 336(2):209 – 234, 2005. Abstract State Machines and High-Level System Design and Analysis.

[7] D. P. Attard and A. Francalanza. A Monitoring Tool for a Branching-Time Logic. In Y. Falcone and C. Sánchez, editors, *Proc. of the 16th International Conference on Runtime Verification (RV)*, volume 10012 of *LNCS*, pages 473–481. Springer, 2016.

[8] D. P. Attard and A. Francalanza. Trace partitioning and local monitoring for asynchronous components. In A. Cimatti and M. Sirjani, editors, *Software Engineering and Formal Methods - 15th International Conference, SEFM 2017, Trento, Italy, September 4-8, 2017, Proceedings*, volume 10469 of *LNCS*, pages 219–235. Springer, 2017.

[9] H. Barringer, Y. Falcone, K. Havelund, G. Reger, and D. E. Rydeheard. Quantified event automata: Towards expressive and efficient runtime monitors. In D. Giannakopoulou and D. Méry, editors, *FM*, volume 7436 of *Lecture Notes in Computer Science*, pages 68–84. Springer, 2012.

[10] H. Barringer, A. Goldberg, K. Havelund, and K. Sen. Rule-based runtime verification. In B. Steffen and G. Levi, editors, *Verification, Model Checking, and Abstract Interpretation*, volume 2937 of *LNCS*, pages 44–57. Springer Berlin Heidelberg, 2004.

[11] E. Bartocci, Y. Falcone, A. Francalanza, and G. Reger. Introduction to runtime verification. In E. Bartocci and Y. Falcone, editors, *Lectures on Runtime Verification - Introductory and Advanced Topics*, volume 10457 of *LNCS*, pages 1–33. Springer, 2018.

[12] G. Boudol and K. G. Larsen. Graphical versus logical specifications. *Theor. Comput. Sci.*, 106(1):3–20, 1992.

[13] J. Bradfield and C. Stirling. Chapter 4 - modal logics and mu-calculi: An introduction. In J. Bergstra, A. Ponse, and S. Smolka, editors, *Handbook of Process Algebra*, pages 293 – 330. Elsevier Science, Amsterdam, 2001.

[14] G. Brat, D. Drusinsky, D. Giannakopoulou, A. Goldberg, K. Havelund, M. Lowry, C. Pasareanu, A. Venet, W. Visser, and R. Washington. Experimental evaluation of verification and validation tools on Martian rover software. *FMSD*, 25(2-3):167–198, 2004.

[15] E. Chang, Z. Manna, and A. Pnueli. Characterization of temporal property classes. In *ALP*, volume 623 of *LNCS*, pages 474–486. Springer-Verlag, 1992.

[16] C. Cini and A. Francalanza. An LTL proof system for runtime verification. In C. Baier and C. Tinelli, editors, *Proc. of the 21st International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 9035 of *LNCS*, pages 581–595. Springer, 2015.

[17] E. M. Clarke, Jr., O. Grumberg, and D. A. Peled. *Model Checking.* MIT Press, 1999.

[18] B. D'Angelo, S. Sankaranarayanan, C. Sánchez, W. Robinson, B. Finkbeiner, H. B. Sipma, S. Mehrotra, and Z. Manna. Lola: Runtime monitoring of synchronous systems. In *TIME*, pages 166–174. IEEE, June 2005.

[19] D. de Frutos-Escrig, C. Gregorio-Rodríguez, M. Palomino, and D. Romero-Hernández. Unifying the linear time-branching time spectrum of process semantics. *Logical Methods in Computer Science*, 9(2), 2013.

[20] N. Decker, M. Leucker, and D. Thoma. jUnitRV - Adding Runtime Verification to jUnit. In *NASA Formal Methods*, volume 7871 of *LNCS*, pages 459–464. Springer, 2013.

[21] A. Desai, T. Dreossi, and S. A. Seshia. Combining model checking and runtime verification for safe robotics. In S. Lahiri and G. Reger, editors, *Runtime Verification*, pages 172–189, Cham, 2017. Springer International Publishing.

[22] A. Francalanza. A Theory of Monitors (Extended Abstract). In *Foundations of Software Science and Computation Structures - 19th International Conference, FOSSACS, Eindhoven, The Netherlands*, volume 9634 of *LNCS*, pages 145–161, 2016.

[23] A. Francalanza. Consistently-detecting monitors. In *28^{th} International Conference on Concurrency Theory (CONCUR)*, volume 85 of *LIPIcs*, pages 8:1–8:19. Schloss Dagstuhl, 2017.

[24] A. Francalanza, L. Aceto, A. Achilleos, D. P. Attard, I. Cassar, D. Della Monica, and A. Ingólfsdóttir. A Foundation for Runtime Monitoring. In *Proc. of the 17th International Conference on Runtime Verification (RV)*, volume 10548 of *LNCS*, pages 8–29. Springer, 2017.

[25] A. Francalanza, L. Aceto, and A. Ingólfsdóttir. On verifying Hennessy-Milner logic with recursion at runtime. In E. Bartocci and R. Majumdar, editors, *Runtime Verification - 6th International Conference, RV 2015 Vienna, Austria, September 22-25, 2015. Proceedings*, volume 9333 of *LNCS*, pages 71–86. Springer, 2015.

[26] A. Francalanza, L. Aceto, and A. Ingólfsdóttir. Monitorability for the Hennessy-Milner logic with recursion. *Formal Methods in System Design*, 51(1):87–116, 2017.

[27] A. Francalanza and A. Seychell. Synthesising Correct Concurrent Runtime Monitors. *Formal Methods in System Design*, pages 1–36, 2014.

[28] K. Kejstová, P. Ročkai, and J. Barnat. From model checking to runtime verification and back. In S. Lahiri and G. Reger, editors, *Runtime Verification*, LNCS, pages 225–240, Cham, 2017. Springer International Publishing.

[29] M. Kim, M. Viswanathan, S. Kannan, I. Lee, and O. Sokolsky. Java-MaC: A Run-Time Assurance Approach for Java Programs. *FMSD*, 24(2):129–155, 2004.

[30] M. Leucker and C. Schallhart. A brief account of Runtime Verification. *JLAP*, 78(5):293–303, 2009.

[31] Z. Manna and A. Pnueli. Completing the Temporal Picture. *TCS*, 83(1):97–130, 1991.

[32] P. O. Meredith, D. Jin, D. Griffith, F. Chen, and G. Roşu. An overview of the MOP runtime verification framework. *STTT*, 14(3):249–289, 2012.

[33] A. Pnueli and A. Zaks. PSL model checking and run-time verification via testers. In J. Misra, T. Nipkow, and E. Sekerinski, editors, *FM 2006: Formal Methods, 14th International Symposium on Formal Methods*, volume 4085 of *LNCS*, pages 573–586. Springer, 2006.

[34] G. Reger, H. C. Cruz, and D. E. Rydeheard. MarQ: Monitoring at runtime with QEA. In C. Baier and C. Tinelli, editors, *Proc. of the 21st International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 9035 of *LNCS*, pages 596–610. Springer, 2015.

[35] R. J. van Glabbeek. The linear time-branching time spectrum II. In *Proc. of the 4th International Conference on Concurrency Theory (CONCUR)*, volume 715 of *Lecture Notes in Computer Science*, pages 66–81. Springer, 1993.

[36] R. J. van Glabbeek. The linear time-branching time spectrum I. In *Handbook of Process Algebra*, pages 3–99. North-Holland/Elsevier, 2001.

[37] S. Varvaressos, D. Vaillancourt, S. Gaboury, A. Blondin Massé, and S. Hallé. Runtime monitoring of temporal logic properties in a platform game. In *Runtime Verification*, volume 8174 of *LNCS*, pages 346–351. Springer, 2013.