Towards A Hybrid Approach to Software Verification (Extended Abstract)

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

[1] ICE-TCS, Reykjavik University, Iceland
dario@ru.is
[2] CS, ICT, University of Malta, Malta
adrian.francalanza@um.edu.mt

Model checking (MC) [6] 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) [10, 14] is a lightweight verification technique aimed at mitigating scalability issues such state explosion problems, 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. 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.*, [2, 3, 8, 9, 12, 13, 16], and has proved effective in various scenarios [4, 7, 17].

Despite its advantages, RV is limited when compared to MC because certain correctness properties cannot be verified at runtime [5, 10, 15]. For instance, MC 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 [5]) 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 [14].

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 [11], one such branching-time logic called μHML [1] is studied from an RV perspective. Figure 1 outlines the logic μHML used and its semantics, defined over a *Labelled Transition System* (LTS), consisting of a set of *states* $s, r \in \textsc{Sta}$, sets of *actions* $\alpha \in \textsc{Act}$, and a *transition relation* between states labelled by actions, $s \xrightarrow{\alpha} r$; as in [1], the semantic definition employs an *environment* from μHML logical variables, $\textsc{Vars}$, to sets of states, $\rho \in (\textsc{Vars} \rightharpoonup \mathcal{P}(\textsc{Sta}))$. One of the main contributions of [11] is the identification of an *expressively maximal, runtime-verifiable subset* of the logic, reported in Figure 1 as the grammar for sHML and cHML; in [11] they show how these classes provide an easy syntactic check for determining whether a property satisfaction (or violation) can be determined using the RV technique.

We building on the findings of [11], 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 *hybrid 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 it *can either* perform action

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

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

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

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 [11]. 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 [11]. We argue that reformulations such as (2) allow for a *hybrid* compositional approach to verification, where part of the property, *e.g.*, the 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.

Preliminary investigations indicate that this decomposition approach applies to arbitrary μHML formulas. We therefore aim to devise *general* analysis techniques that reformulate *any* μHML formula into either conjunctions or disjunctions, *i.e.*, $\varphi_{\text{RV}} \wedge \varphi_{\text{MC}}$ or $\varphi_{\text{RV}} \vee \varphi_{\text{MC}}$, 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 hybrid 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 $\varphi_{\text{MC}} = \langle a\rangle[c]\mathsf{ff}$ 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 hybrid 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.* $\varphi_{\mathrm{MC}} = \langle a\rangle[c]\mathsf{ff}$.

References

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

[2] 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. Mry, editors, *FM*, volume 7436 of *Lecture Notes in Computer Science*, pages 68–84. Springer, 2012.

[3] 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.

[4] 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.

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

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

[7] C. Colombo and G. J. Pace. Fast-forward runtime monitoring - an industrial case study. In *Runtime Verification*, volume 7687 of *LNCS*, pages 214–228. Springer, 2012.

[8] 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.

[9] 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.

[10] Y. Falcone, J.-C. Fernandez, and L. Mounier. What can you verify and enforce at runtime? *STTT*, 14(3):349–382, 2012.

[11] A. Francalanza, L. Aceto, and A. Ingolfsdottir. On verifying Hennessy-Milner logic with recursion at runtime. In *Runtime Verification*. Springer, 2015. (to appear).

[12] A. Francalanza and A. Seychell. Synthesising Correct Concurrent Runtime Monitors. *FMSD*, pages 1–36, 2014.

[13] 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.

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

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

[16] 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.

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