Towards A Hybrid Approach to Software Verification (Extended Abstract)

Dario Della Monica\textsuperscript{1} and Adrian Francalanza\textsuperscript{2}

\textsuperscript{1} ICE-TCS, Reykjavik University, Iceland
\texttt{dario@ru.is}

\textsuperscript{2} CS, ICT, University of Malta, Malta
\texttt{adrian.francalanza@um.edu.mt}

Model checking (MC) \cite{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) \cite{10, 14} is a lightweight verification technique aimed at mitigating scalability issues such as 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., \cite{2, 3, 8, 9, 12, 13, 16}, and has proved effective in various scenarios \cite{4, 7, 17}.

Despite its advantages, RV is limited when compared to MC because certain correctness properties cannot be verified at runtime \cite{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 \cite{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 \cite{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 \cite{11}, one such branching-time logic called $\mu$HML \cite{1} is studied from an RV perspective. Figure 1 outlines the logic $\mu$HML used and its semantics, defined over a Labeled Transition System (LTS), consisting of a set of states $s, r \in \text{STA}$, sets of actions $\alpha \in \text{ACT}$, and a transition relation between states labelled by actions, $s \xrightarrow{\alpha} r$; as in \cite{1}, the semantic definition employs an environment from $\mu$HML logical variables, VARS, to sets of states, $\rho \in (\text{VARS} \rightarrow \mathcal{P}($\text{STA}$))$. One of the main contributions of \cite{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 \cite{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 \cite{11}, with the aim of extending the applicability of RV to a larger class of $\mu$HML 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 $\mu$HML 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
Towards A Hybrid Approach to Software Verification

Syntax

\[ \varphi, \phi \in \muHML := \begin{cases} \text{tt} & (\text{truth}) \\ \text{ff} & (\text{falsehood}) \\ \varphi \lor \phi & (\text{disjunction}) \\ \varphi \land \phi & (\text{conjunction}) \\ \langle \alpha \rangle \varphi & (\text{possibility}) \\ [\min X, \varphi] & (\text{min. fixpoint}) \\ [X, \varphi] & (\text{rec. variable}) \end{cases} \]

Semantics

\[
\begin{align*}
\text{[tt, } \rho \text{]} & \overset{\text{def}}{=} \text{STA} \\
[\varphi_1 \lor \varphi_2, \rho] & \overset{\text{def}}{=} [\varphi_1, \rho] \cup [\varphi_2, \rho] \\
[\langle \alpha \rangle, \varphi, \rho] & \overset{\text{def}}{=} \{ s \mid \exists r.s \xrightarrow{r} r \text{ and } r \in [\varphi, \rho] \} \\
[\min X, \varphi, \rho] & \overset{\text{def}}{=} \bigcap \{ S \in \text{STA} \mid [\varphi, \rho][X \mapsto S] \subseteq S \} \\
[X, \varphi, \rho] & \overset{\text{def}}{=} \rho(X)
\end{align*}
\]

Monitorable Fragments

\[
\begin{align*}
\theta, \theta & \in sHML := \text{tt} \\
\pi, \varpi & \in cHML := \text{tt}
\end{align*}
\]

Figure 1: \(\muHML\) Syntax and Semantics

\(b\), subformula \(\langle b \rangle \text{tt}\), or else can never perform action \(c\), subformula \(\langle c \rangle \text{ff}\).

\[
\langle a \rangle (\langle b \rangle \text{tt} \lor \langle c \rangle \text{ff}) \tag{1}
\]

According to Figure 1, \(\langle a \rangle (\langle b \rangle \text{tt} \lor \langle c \rangle \text{ff})\) turns out not to be runtime-verifiable because of the subformula \(\langle c \rangle \text{ff}\); intuitively, whereas a system execution exhibiting action \(a\) followed by action \(b\) suffices to prove that the system satisfies \(\text{(1)}\), 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 \(\text{(1)}\). However, property \(\text{(1)}\) can be expressed as the (logically equivalent) formula

\[
(\langle a \rangle (\langle b \rangle \text{tt}) \lor (\langle a \rangle \langle c \rangle \text{ff})) \tag{2}
\]

whereby we note that the subformula \(\langle a \rangle (\langle b \rangle \text{tt})\) is runtime verifiable, according to \(\text{(1)}\). We argue that reformulations such as \(\text{(2)}\) allow for a hybrid compositional approach to verification, where part of the property, \(e.g.,\), the subformula \(\langle a \rangle \langle c \rangle \text{ff}\), can be checked prior system deployment using MC, and the remaining part of the property, \(e.g.,\), \(\langle a \rangle (\langle b \rangle \text{tt}),\) can be runtime-verified during system execution.

Preliminary investigations indicate that this decomposition approach applies to arbitrary \(\muHML\) formulas. We therefore aim to devise general analysis techniques that reformulate any \(\muHML\) formula into either conjunctions or disjunctions, \(i.e.,\), \(\varphi_{\text{RV}} \land \varphi_{\text{MC}}\) or \(\varphi_{\text{RV}} \lor \varphi_{\text{MC}}\), where \(\varphi_{\text{RV}}\) and \(\varphi_{\text{MC}}\) 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 \(\text{(2)}\), the model-checked subformula \(\varphi_{\text{MC}} = \langle a \rangle \langle c \rangle \text{ff}\) is smaller than the full formula \(\text{(1)}\), since we would be offloading a degree of verification onto the runtime phase when runtime-verifying for
\( \varphi_{RV} = (a)(b)tt \). Moreover, for disjunction decompositions such as \( \varphi_{MC} \), the satisfaction of \( \varphi_{MC} \) prior to deployment obviates the need for any runtime analysis, minimising runtime overheads (a dual argument applies for conjunction decompositions and \( \varphi_{MC} \) 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 \( \varphi_{RV} = (a)(b)tt \) during its pilot launch, acting as a vetting phase: if \( \varphi_{RV} \) 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. \( \varphi_{MC} = (a)[c]ff \).

References


