Monitoring Interfaces for Faults

Aleksandr Zaks

RV’05 - Fifth Workshop on Runtime Verification

Joint work with:

Amir Pnueli, Lenore Zuck
Motivation

- Consider two components interacting with each other. We will refer to them as module and interface.

- We assume that we have all the implementation details of the module while only partial (incomplete) specification is available for the interface.

- As usual, we want to make sure our system comprised of the above components is a correct one.

A possible scenario is when an in-house application uses some third-party components.
Compile Time Approaches

- The interface specification being incomplete means that the interface displays only a strict subset of the behaviors allowed by the its specification.

- Therefore, while it might perfectly ok to use the external component (interface), applying formal methods, like model checking, will usually result in false negatives – a system incorrectly declared faulty.

- Moreover, in our work, we also consider a case when the interface specification is not only partial, but simply wrong. Meaning the interface may not satisfy its own specification. A situation may arise when a third-party component is "upgraded" without notifying the end users.

To conclude: it is impossible to rely solely on compile-time verification when the information about parts of the program is either missing or incorrect.
Towards Run-Time Monitoring

Instead of formal verification, a user may prefer to employ some form of run-time monitoring

- A straightforward approach of run-time monitoring is to check that a currently observed execution trace satisfies the system specification. This approach avoids all the problems mentioned before by simply discarding all the information about the module and the interface.

- The main drawback is of course the loss of precision – too many bugs are left undetected. Assuming traditional run-time monitoring is relatively fast, it might be desirable to trade efficiency for better precision.

- In particular, we propose to use all the information available: implementation details of the module, specification of the interface, run-time information, and of course the overall specification of the system.
Sketch of the Proposed Solution.

- View the module and the interface as players in a 2-player game.
- The interface has a winning strategy if it can guarantee that no matter what module does, the overall specification $\Phi$ is met.
- Interface can follow any strategy consistent with its own specification $\Phi_I$.
- At every point of the execution we check if the environment can win, assuming the strategies available for both the module and the interface are restricted according to the history of a computation.
- Since one can always view a particular implementation of the interface as a strategy, a fault-free interface can always win against a fault-free module. Therefore, we generate an alert when no suitable strategy exists.
Following [dAH1,dAH2] we define a (two-player) game $G = (S, A, \Gamma_1, \Gamma_2, \delta)$ to consist of:

- a finite set $S$ of states;
- a finite set $A$ of actions;
- action assignment functions $\Gamma_1, \Gamma_2: S \rightarrow 2^A \setminus \emptyset$ that define, for each state, a non-empty set of actions available to player 1 and player 2 respectively;
- a transition function $\delta: S \times A \times A \rightarrow S$ that associates with each state $s$ and each pair of actions $(a, b) \in \Gamma_1(s) \times \Gamma_2(s)$ a successor state $\delta(s, a, b)$;

Intuitively, from each state, each of the players chooses an action, which are taken simultaneously. The two actions define the next state of the system.
Game Strategies and Runs

Given a game $G$ as above, and $i \in \{1, 2\}$,

- A **player-$i$ strategy** is a function $\xi_i : S^+ \rightarrow A$ that maps, for every nonempty finite sequence $\sigma \in S^+$, a single action that is consistent with $\Gamma_i$, i.e., for every $\sigma \in S^*$ and $s \in S$, $\xi_i(\sigma; s) \in \Gamma_i(s)$.

- The set of strategies for player-$i$ is denoted by $\Xi_i$.

- A **run** $r$ of the game structure $G$ is a nonempty, possibly infinite sequence $s_0(a_0, b_0)s_1(a_1, b_1)s_2 \ldots$ of alternating states and action pairs such that for every $j \geq 0$, $a_j \in \Gamma_1(s_j)$, $b_j \in \Gamma_2(s_j)$, and $s_{j+1} = \delta(s_j, a_j, b_j)$.

- For a run $r : s_0(a_0, b_0)s_1(a_1, b_1)s_2 \ldots$, we refer to the state sequence $\sigma(r) : s_0, s_1, s_2, \ldots$ as the **history induced by** $r$.

- Given a pair of strategies $\xi_1 \in \Xi_1$ and $\xi_2 \in \Xi_2$ and a state $s \in S$, the **outcome of the strategies from** $s$, $H_{\xi_1, \xi_2}(s)$, is the history induced by the run whose initial state is $s$ and whose actions are consistent with the strategies.
Winning for an LTL Objective

- Let \( h : s_0, s_1, \ldots, s_k = s \) be a finite history and \( \Psi \) a linear temporal logic formula over \( S \).

- History \( h \) is said to be a winning history for player-\( i \), \( i \in \{1, 2\} \), with respect to objective \( \Psi \) in \( G \) if player-\( i \) has a strategy \( \xi_i \in \Xi_i \) such that for all strategies \( \xi_{3-i} \in \Xi_{3-i} \), \( h \cdot H_{\xi_1,\xi_2}(s) \models \Psi \).

- A suitable strategy \( \xi_i \) is a winning player-\( i \) strategy for \( \Psi \) from \( h \) in \( G \). In case a winning history \( h \) consists of the single state \( s \), we refer to \( s \) as a player-\( i \) winning state.
Example of A Game Structure (1/2)

- Blue/Red means player 1/2 can force to take the transition.
- Our objective $\Psi = \square ((s_1 \rightarrow \bigcirc \bigcirc s_4) \land (s_2 \rightarrow \bigcirc \bigcirc s_5))$.
- Let’s identify which states are winning for which of the players.
Blue states are *player-1 winning*. Magenta states are winning for both players.

Our objective $\Psi = \Box ((s_1 \rightarrow \bigcirc \bigcirc s_4) \land (s_2 \rightarrow \bigcirc \bigcirc s_5))$.

Consider two histories $h : s_0, s_1, s_3, s_4$ and $h' : s_0, s_2, s_3, s_4$. Clearly, $h$ is a winning history for player-2, while $h'$ is not. Therefore, whenever the game reaches $s_4$ we cannot immediately decide whether we should raise an alarm.

We say that a game $G$ is *partitionable* when all the states are winning regardless of the history of the game. Note that non-partitionable games are not suitable for run-time monitoring since we must take into account the history of the game, which can be arbitrarily long.
Universal Liveness

- An LTL formula $\Omega$ represents a universal liveness property if the following holds:

  For every $\sigma_1 \in \Sigma^*$ and $\sigma_2 \in \Sigma^\omega$, $\sigma_1 \cdot \sigma_2 \models \Omega$ iff $\sigma_2 \models \Omega$.

- It can be shown that if the objective $\Psi$ of game $G$ is a universal liveness property, then $G$ is a partitionable game.

- Universal liveness should not be confused with the concept of memoryless strategies. Consider the objective $\Psi = [] [] s_1 \land [] [] s_2$ and the game below.

```
\begin{center}
\begin{tikzpicture}
  \node (s0) at (0,0) {$s_0$};
  \node (s1) at (1,1) {$s_1$};
  \node (s2) at (1,-1) {$s_2$};
  \path[->]
  (s0) edge node [above] {$\langle c, a \rangle$} (s1)
  (s1) edge node [above] {$\langle c, c \rangle$} (s2)
  (s2) edge node [above] {$\langle c, b \rangle$} (s0)
  (s0) edge node [above] {$\langle c, c \rangle$} (s1)
\end{tikzpicture}
\end{center}
```
Constructing a Partitionable Game

- Compose non-partitionable game $G$ with a deterministic Büchi/Rabin automaton for the objective of the game $G$ to obtain a new game $G'$.  

- The objective of the composed game $G'$ is the acceptance condition of the automaton.  

- Since the acceptance condition only depends on the states that appear infinitely often, the new objective represents a universal liveness property. Therefore, the composed game must be partitionable.
We take a *fair discrete system* (FDS), which is a variant of *fair transition system* [MP95], as our computational model. Under this model, a system $M : \langle V, W, \Theta, \rho, J \rangle$ consists of the following components:

- $V$: A finite set of *system variables*. A state of the system $S$ provides a type-consistent interpretation of the system variables $V$. For a state $s$ and a variable $v \in V$, we denote the value assigned to $v$ by the state $s$ by $s[v]$. Let $\Sigma$ denote the set of all states over $V$. We assume that $\Sigma$ is finite.

- $W \subseteq V$: A subset of *owned* variables which only the system itself can modify. All other variables can also be modified by the environment.

- $\Theta$: The *initial condition*. A state is defined to be *initial* if it satisfies the assertion (state formula) $\Theta$.

- $\rho(V, V')$: The *transition relation*.

- $J$: A set of *justice* (*weak fairness*) requirements (assertions). For every $J \in J$, $\sigma$ contains infinitely many occurrences of $J$-states.
Given:

- an FDS $M = (V_M, W, \Theta, \rho, J)$ that corresponds to some SPL module (our system)
- a goal specification $\Phi$
- an interface (off-the-shelf component) specification $\Phi_I$

We define a game $G = (S, A, \Gamma_1, \Gamma_2, \delta)$ between the module $M$ and the interface as follows:

- Let $V = V_M \cup \{\text{turn}\}$, where $\text{turn} \in \{1, 2\}$ is a variable not in $V_M$.
- $S$ is the set of all type-consistent interpretations over $V$.
- We take $A$, the set of $G$'s actions, to be $S$. 
In the game $G$, player 1 (the module) can take any step consistent with $\rho$, non-deterministically setting $\text{turn}$, thus deciding whether or not it wishes to take another step. Player 2 (the interface) can change one variable that is not owned by $M$, but it must let Player 1 take the next step.

Formally:

- $\Gamma_1(s) = \{s' | \rho(s, s')\}$
- $\Gamma_2(s) = \{s' | s'[\text{turn}] = 1 \land (\bigvee_{v \in V_M \setminus W} \text{pres}(V \setminus \{v, \text{turn}\}))\}$
- $\delta(s, a_1, a_2) = a_s[\text{turn}]$

Thus, each player expresses its preference for the state it would like to see next. The final decision is determined as the choice of the player whose turn it is.
The objective of the game is defined by

$$\Psi = (\Phi \land \Phi_I) \lor \lozenge \Box (\text{turn} = 1 \lor \bigvee_{J \in J} \neg J)$$

Thus, the objective of the game (wishing the interface to win) is to ensure that all computations either

- satisfy both $\Phi$ and $\Phi_I$, or
- violate one of the justice requirements, or
- allow the interface to take only finitely many steps.
Proposed Solution

- Construct a partitionable game $G$ given the module, interface specification $\Phi_I$, and goal specification $\Phi$.

- Compute the set $Win$ of winning states for the interface.

- Monitor the execution of the system to make sure the game stays within the set $Win$. 
Monitoring Example

\[ \begin{align*}
\text{shared boolean } & \ flag_1, flag_2 = 0 \\
\text{local boolean } & \ error = 0 \\
\ell_0 : \ & \text{while } \neg error \\
& \left[ \begin{array}{l}
\ell_1 : \ flag_1 := 0 \\
\ell_2 : \ await flag_1 \\
\ell_3 : \text{if } flag_2 \text{ then } error := 1
\end{array} \right]
\ell_4 : \\
\end{align*} \]

\[ \begin{align*}
\text{shared boolean } & \ flag_1, flag_2 = 0 \\
m_0 : \ & \text{loop forever do} \\
& \left[ \begin{array}{l}
m_1 : \ await \neg flag_1 \\
m_2 : \ flag_2 := 1 \\
m_3 : \ flag_2 := 0 \\
m_4 : \ flag_1 := 1 \\
m_5 : \\
\end{array} \right]
\end{align*} \]

(a) Module \( M \)

(b) Interface \( I^* \)

- Assume the system specification is \( \Phi : \Box \ Diamond \ at_0 \land \Box \ Diamond \ flag_2 \) and the interface specification is the trivial \( \Phi_I : 1 \).

- Consider a state \( at_{\ell_2} \land flag_1 \land flag_2 \). It is easy to verify that it is a losing state for the interface, assuming it is the module’s turn. Consequently, our monitor will raise an alarm when such state is reached.

- The alarm is really an \textit{early} warning. Although the violation is unpreventable (by the interface), it would take infinitely many steps to confirm the violation.
Main Features of our Monitors.

- Our approach is sound. An alert from the monitor implies the existence of a bug. However, it does not mean a violation will occur in the observed trace.

- Alerts are really early warnings. However, we can change the game slightly (let the interface decide for the module) to detected the situations when a violation must occur in the observed trace.

- Our method is not complete. Too much power is given to the interface. In particular, the interface can observe local variables of the module. Therefore, sometimes we may fail to infer a bug.

- It should be stressed that we always generate an alert as soon as a violation happens in the observed trace.
Efficiency vs Precision.

System:

- **Interface** is empty – model checking.
- **Module** is empty – traditional run-time monitoring.
- Both parts are present – our framework.
- We assume the split(abstraction) is defined by the user and the presence of third-party components. Our approach, with some exceptions, yields the most precise results for the given abstraction.
- Although our approach will handle the extreme cases it might be more efficient to use a dedicated solution. For example, in case everything is treated as an interface one may want to use approach described in ”Efficient Monitoring of Omega-Languages” by Marcelo d’Amorim and Grigore Rosu.
Computing Winning States (1/2).

- The **objective** of the game is defined by

  $\Psi = (\Phi \land \Phi_I) \lor \Diamond \Box (\text{turn} = 1 \lor \bigvee_{J \in J} \neg J)$

- After composition with a Büchi automaton for $\Phi \land \Phi_I$

  $\Psi' = \Box \Diamond F \lor \Diamond \Box (\text{turn} = 1 \lor \bigvee_{J \in J} \neg J)$

- Consider an LTL objective $\Phi$ in the form:

  $\Box \Diamond p \lor \bigvee_{i=1}^{n}(\Diamond \Box r_i)$,

where $p, r_1, \ldots, r_n$ are assertions. Clearly, the objective of $\Psi'$ has such a form. Let $Win$ be a set of player-2 winning states for $\Phi$. 
Computing Winning States (2/2).

- Following [kesten03cav], we can compute $Win$ as follows:

$$Win = \nu Z. \mu Y. (\bigvee_{i=1}^{n} \nu X. (p \land \Box Z) \lor \Box Y \lor (r_i \land \Box X)),$$

where

$$[[\Box f]]_{G}^{e} = \{ s \in S \mid \exists b \in \Gamma_2(s). \forall a \in \Gamma_1(s). \delta(s, a, b) \in [[[f]]_{G}^{e}\}.$$  

- That is, $[[\Box f]]_{G}^{e}$ characterizes the set of states from which player-2 can force the game to move to a state belonging to $[[f]]_{G}^{e}$ in one step.

- The above formula can be evaluated symbolically, requiring at most $|S|^2$ steps, in spite of the fact that it has three alternating fix-point operators [LBC94].