Runtime verification for LTL

Doina Bucur

RuG

Jan 2013
In this lecture...

Runtime verification: overview
System logging
From temporal specification to monitors
Runtime checking

We see a form of model checking suitable to verify not a system design, but an execution trace of a system, at system runtime. This is akin to testing in purpose, but allows for formal temporal specifications and guarantees. This method is called runtime verification, or simply monitoring, and is computationally lightweight.

Runtime verification works as follows: (1) the system to be monitored is logged for runtime events (which model atomic propositions from $AP$); (2) the runtime trace of runtime events is a finite word over $AP$; (3) ...which is checked for inclusion in the language generated by a LTL formula $f$, as for explicit-state model checking.
A system execution

...is formally a simple, linear transition system (as in Lectures 2/3):

\[
\begin{array}{cccccccc}
\circ & \rightarrow & \circ & \rightarrow & \cdots & \rightarrow & \circ & \rightarrow \\
S_0 & & S_1 & & S_2 & & \cdots & & S_{k-1} & & S_k & & \cdots
\end{array}
\]

where:

- labels are attached to transitions (as in Büchi automata)
- \(s_0\) is the first system state under monitoring
- \(s_k\) is the “current” or “now” system state (i.e., after the latest execution step)
- the path \(s_0s_1s_2\ldots s_k\) is necessarily finite, and increasing
- states following \(s_k\) are unknown (and the entire execution possibly infinite)
Runtime verification: overview

system execution
(a single word over AP)

temporal property translated into
a (new type of) Büchi automaton

Runtime verification amounts to checking the inclusion of the execution word into the property language.
Building blocks for a runtime verification tool

- A set of atomic propositions AP of interest.
- A temporal specification over AP, and its translation into an appropriate automata form.
  - This is usually called a monitor.
  - To express this automaton, I show you a new type of Büchi automata called Transition-based Generalized Büchi Automata (TGBA).
- A model, i.e., a linear transition system (with transition labels from AP) obtained from the system execution.
- An implemented verification algorithm to check whether the model is “included” in the specification. This is extremely simple: it amounts to executing the monitor over the model.

Crucial fact: all of the above reside and run on the system under test. This adds memory overhead (overall linear in the size of the monitor) and computational overhead.

In particular, the monitor must be small.
A real-life framework: Java Path Explorer

A runtime verification tool for Java (from NASA Ames)\textsuperscript{12}.


\textsuperscript{2}Maude is an LTL checker.
A real-life framework: Microcontroller monitoring

A runtime verification tool for binary code.

## Runtime verification vs. model checking

<table>
<thead>
<tr>
<th>Model checking</th>
<th>Runtime verification</th>
</tr>
</thead>
<tbody>
<tr>
<td>acts at design- or compile-time</td>
<td>acts at runtime</td>
</tr>
<tr>
<td>prevents errors</td>
<td>detects errors</td>
</tr>
<tr>
<td>checks many (infinite) executions</td>
<td>checks one finite execution</td>
</tr>
<tr>
<td>is computationally heavyweight</td>
<td>is computationally lightweight</td>
</tr>
<tr>
<td>does not modify the system</td>
<td>adds logging overhead to the system</td>
</tr>
<tr>
<td>both support temporal logics</td>
<td></td>
</tr>
<tr>
<td>can check for safety and liveness</td>
<td>can check for safety</td>
</tr>
<tr>
<td>(often uses past-time temporal logics)</td>
<td></td>
</tr>
<tr>
<td>is a language-inclusion problem</td>
<td>is a word-inclusion problem</td>
</tr>
<tr>
<td>a check can be complete</td>
<td>a check is incremental</td>
</tr>
</tbody>
</table>
Usability of runtime verification

- Research is mainly focused on efficient algorithms for the real-time detection of violations of specifications.
- When a violation is detected over a real system, it is expected that predefined code is executed, to, e.g., notify users about the error, contain the error, or correct the error.
- We don’t discuss matters of error correction, but focus on error detection.
System instrumentation with logging

Aim: obtain the model of the execution.

Method:

- Decide on the relevant AP, e.g.
  \[ p := \text{function()} \text{ is called} \]
  \[ q := (\text{memory}\_\text{location} == 0xbeef) \]
  \[ t := (\text{ncrit} > 1) \]
- Engineer the system with added code so that events are triggered when the boolean values of \( p, q, r \) change.

This is sufficient to infer the required transition system.
Transition-based generalized Büchi automata (TGBA)

You have seen that central to explicit-state model checking are Büchi automata, a form of ω-automata where acceptance equals traversing infinitely often a set of accepting states.

It was found that transition-based automata, where acceptance is defined in term of transitions, generally leads to smaller automata (and monitor size is important; it determines the amount of runtime overhead introduced by the runtime verification).
TGBA: definition

A **Transition-based Generalized Büchi Automaton (TGBA)** over the alphabet $\Sigma$ is a Büchi automaton with labels on transitions, and acceptance conditions also on transitions\(^4\).

**Definition (TGBA)**

A TGBA is a tuple $\mathcal{A} := (\Sigma, Q, \Delta, q_0, F)$ where

- $\Sigma$ is a finite alphabet of symbols.
- The state space $Q$ is finite.
- The initial state is $q_0 \in Q$; there are no final states.
- $F$ is a finite set of acceptance conditions.
- $\Delta$ is the transition relation, $\Delta \subseteq Q \times 2^\Sigma \setminus \emptyset \times 2^F \times Q$.

A TGBA can be constructed for a given LTL property $f$ such that it accepts exactly the temporal words described in $f$.

\(^4\)The TGBAs here were translated from LTL using the SPOT tool [http://spot.lip6.fr/ltl2tgba.html](http://spot.lip6.fr/ltl2tgba.html). For a translation algorithm, see their paper.
TGBA: (in)finite acceptance

An (in)finite word $\nu$ over the alphabet $\Sigma$ is accepted by $A$ if:

- there exists an (in)finite sequence of transitions from $\Delta$, starting at $q_0$, such that $\forall i \geq 0$, $\nu[i]$ is included in the $i$th transition label;
- the sequence of transitions traverses each acceptance condition (in)finitely often.

For runtime verification, only finite acceptance is relevant. Most times, there are no acceptance conditions: all transitions are accepting.
Monitors: examples (safety)

\[ \mathcal{G}_p \]
no acceptance condition (all transitions accepting)

\[ \mathcal{G}(q \rightarrow \mathcal{G}(p)) \]
“p is true after q” (all transitions accepting)

“transitions to p-states occur at most twice before r” (all transitions accepting)
Monitors: examples (safety)

\[ a U (b U (c U (d U (e U (f U (g U (h U (i U (j U k))))))))) \]

an Until 10-chain

no acceptance condition (all transitions accepting)

(expressing this monitor takes 24k of embedded C code)
Monitors: why not also liveness?

Liveness properties are those whose counterexamples are necessarily infinite.

Thus, a liveness property cannot be violated over a finite execution.

The monitor for a liveness property:

\[ Fp \]
(all transitions accepting)
Runtime checking

system execution
(a single word over AP)

Without acceptance conditions, the execution word is accepted when there exists a transition in the monitor for each symbol in the word.

What is the complexity of this check?
Monitors: deterministic or nondeterministic?

The example monitors here are deterministic.

Ideally, the monitors should be **deterministic**: a deterministic monitor has constant computational complexity for checking, instead of linear in the size of the automaton.

However, you’ve only seen Büchi automata being generated (e.g., by Spin) in nondeterministic form. Any nondeterministic automaton over finite words can be determinized; the final step is to do some pruning to eliminate accepting paths for infinite words.