Runtime verification for LTL

[Automated Reasoning, 2015/2016 1b — Lecture 7]

Doina Bucur

d.bucur@rug.nl

Jan 2016

In this lecture...

Runtime verification: overview

From temporal specification to monitors

The checking algorithm: an acceptance check

We see a form of model checking suitable to verify not a system design, but an execution trace of a system. This is akin to *testing*, 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 atomic propositions from AP are now runtime events; (2) the resulting trace of runtime events is a finite word over AP; (3) this word is checked for inclusion in the language generated by a LTL formula f, as usual in automata-based model checking.

Use of runtime verification

- At **each new step** in the system execution: says whether the system satisfies or not the specification... so far.

- Research is mainly focussed on efficient algorithms for the real-time detection of violations of **safety** 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.

Building blocks for automata-based runtime verification

- 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). Other, slightly different types of automata, can also do the job.
- 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 because of the linearity of the model: it amounts to *executing* the monitor over the model.

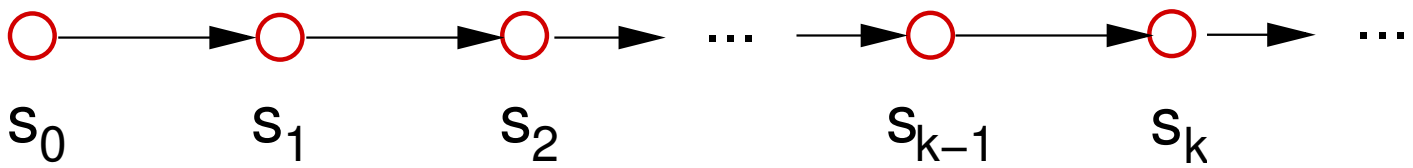
Crucial fact: all of the above reside and run on the system under test. This adds memory overhead (linear in the size of the monitor) and some computational overhead. The monitor must be kept small.

Runtime verification vs. model checking

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

A system execution

...is formally a simple, linear automaton:



where:

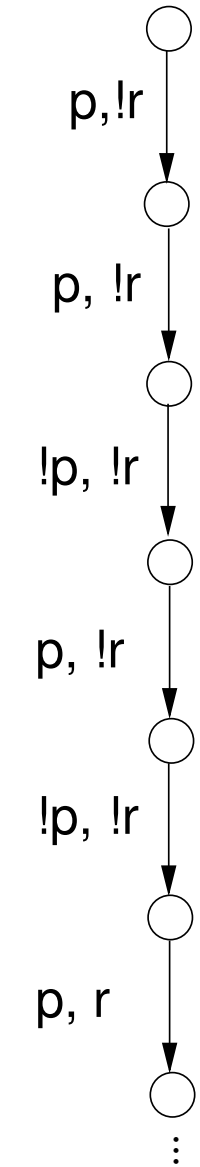
- labels may be attached to transitions (as in Büchi automata) or, equivalently, to states (as in Kripke structures)
- s_0 is the first system state under monitoring
- s_k is the "current" or "now" system state (i.e., after the latest execution step)
- the path $s_0 s_1 s_2 \ldots s_k$ is necessarily finite, and increasing
- states following s_k are unknown

Model the system via logging

Aim: obtain a formal, linear model of the execution.

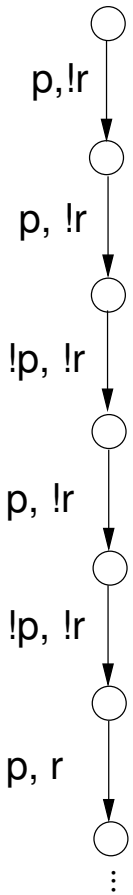
Method:

- Decide on the relevant AP set, e.g.:
  $p :=$ `process0 is at line 5`
  $q :=$ `(pointer == 0xbeef)`
  $r :=$ `(ncrit > 1)`
- This AP is dictated by the specification(s) to be checked.
- Engineer the existing system (written in C, Java bytecode, binary, etc.) with added logging code, so that the boolean variables p, q, r evolve their values correctly. (These changes are called *runtime events*.)



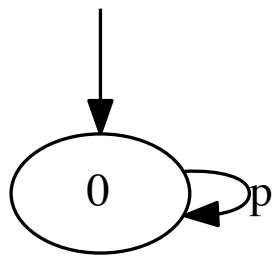
This is sufficient to infer the required model.

Runtime verification: overview



system execution

(a single word over $AP = \{p, r\}$)



positive **temporal property** $\mathbf{G}p$ as a (new type of) Büchi automaton

(all transitions here are accepting)

Runtime verification checks the inclusion of the system execution (a word) into the positive temporal property (a language) — just like static model checking. Only the definition of acceptance changes.

The monitors: Transition-based generalized Büchi automata (TGBA)

You have seen that central to automata-based 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[1] (and monitor size is important; it determines the amount of runtime overhead introduced by the runtime verification).

[1] As per *From States to Transitions: Improving translation of LTL formulae to Büchi automata*. Dimitra Giannakopoulou and Flavio Lerda, FORTE 2002. Online at `http://ti.arc.nasa.gov/m/profile/dimitra/publications/forte02.pdf`.

TGBA: definition

A **Transition-based Generalized Büchi Automaton (TGBA)** over the alphabet Σ is a Büchi automaton with labels on transitions, and acceptance conditions also on transitions rather than states[2].

Definition (TGBA)

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

- Σ is a finite alphabet of symbols.
- The state space Q is finite.
- The initial state is $q_0 \in Q$; **there are no accepting states**.
- F is a finite set of acceptance labels for transitions.
- Δ is the transition relation, $\Delta \subseteq Q \times 2^{\Sigma} \setminus \emptyset \times 2^{F} \times Q$.

[2]The TGBAs here were translated from LTL using the SPOT tool `http://spot.lip6.fr/ltl2tgba.html`. For a translation algorithm, see their paper.

TGBA: finite acceptance

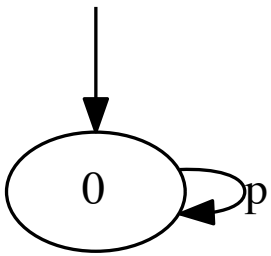
A finite word v over the alphabet Σ is **accepted** by $\mathcal{A}$ if:

- there exists a sequence of transitions from Δ, starting at q_0, such that $\forall i \geq 0$, $v[i]$ is included in the ith transition label
- the sequence of transitions traverses an accepting transition continuously.

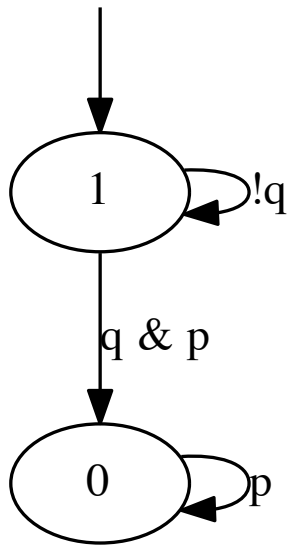
In many practical cases, all cycles are accepting.

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

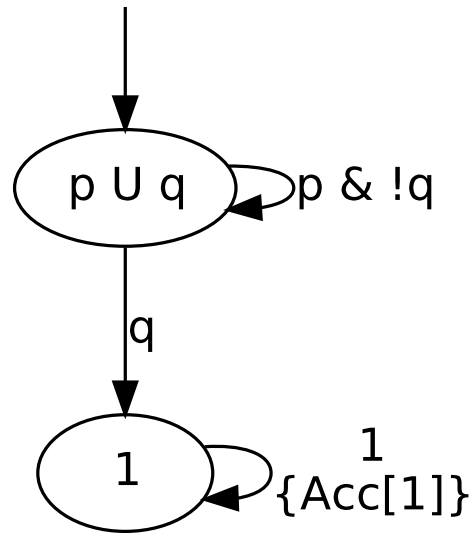
Monitors: examples (safety)



$\mathbf{G}p$
no acceptance condition (all transitions accepting)

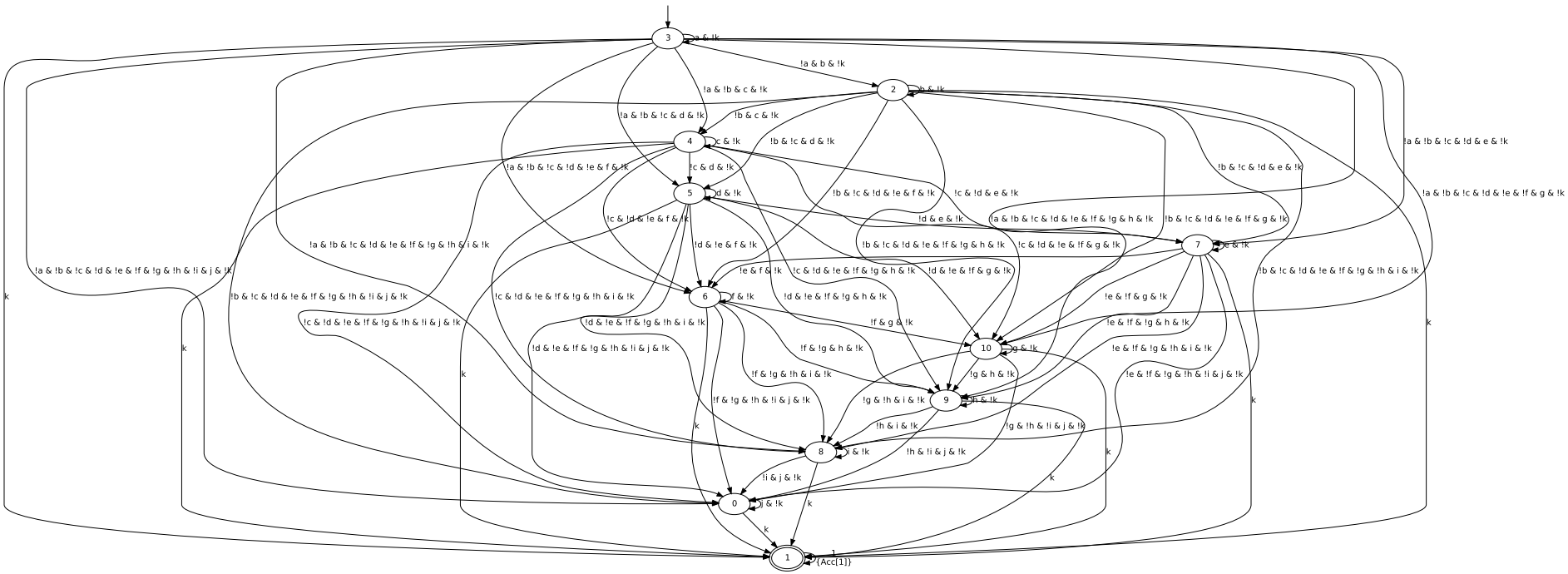


$\mathbf{G}(q \rightarrow \mathbf{G}p)$
"p is true globally after q"
(all transitions accepting)



$p\mathbf{U}q$
(1 acceptance condition: Acc[1])

Monitors: examples (safety)



$$a\mathbf{U}(b\mathbf{U}(c\mathbf{U}(d\mathbf{U}(e\mathbf{U}(f\mathbf{U}(g\mathbf{U}(h\mathbf{U}(i\mathbf{U}(j\mathbf{U}k)))))))))$$

an Until 10-chain

(1 acceptance condition: Acc[1])

(expressing this monitor takes 24k of executable code, compiled from C)

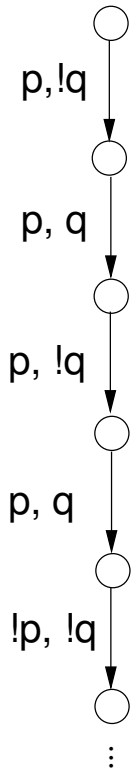
Monitors: why not also liveness?

Liveness properties are those whose counterexamples are necessarily infinite.

Thus, a **purely** liveness property cannot be violated (but can be passed!) over a finite execution.

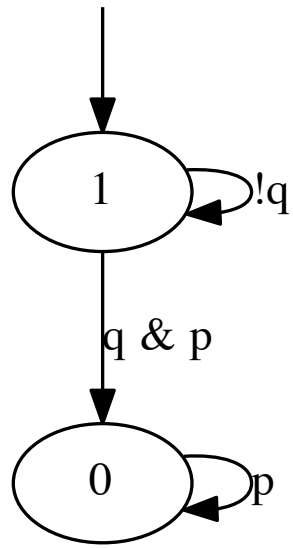
The only results of runtime verification for liveness are either *verification successful* or *undetermined*.

The checking algorithm



system execution

(a single word over AP)



temporal property, $\mathbf{G}(q \rightarrow \mathbf{G}(p))$

(all transitions accepting)

What is the **time complexity** of the acceptance check?

Monitors: deterministic or nondeterministic?

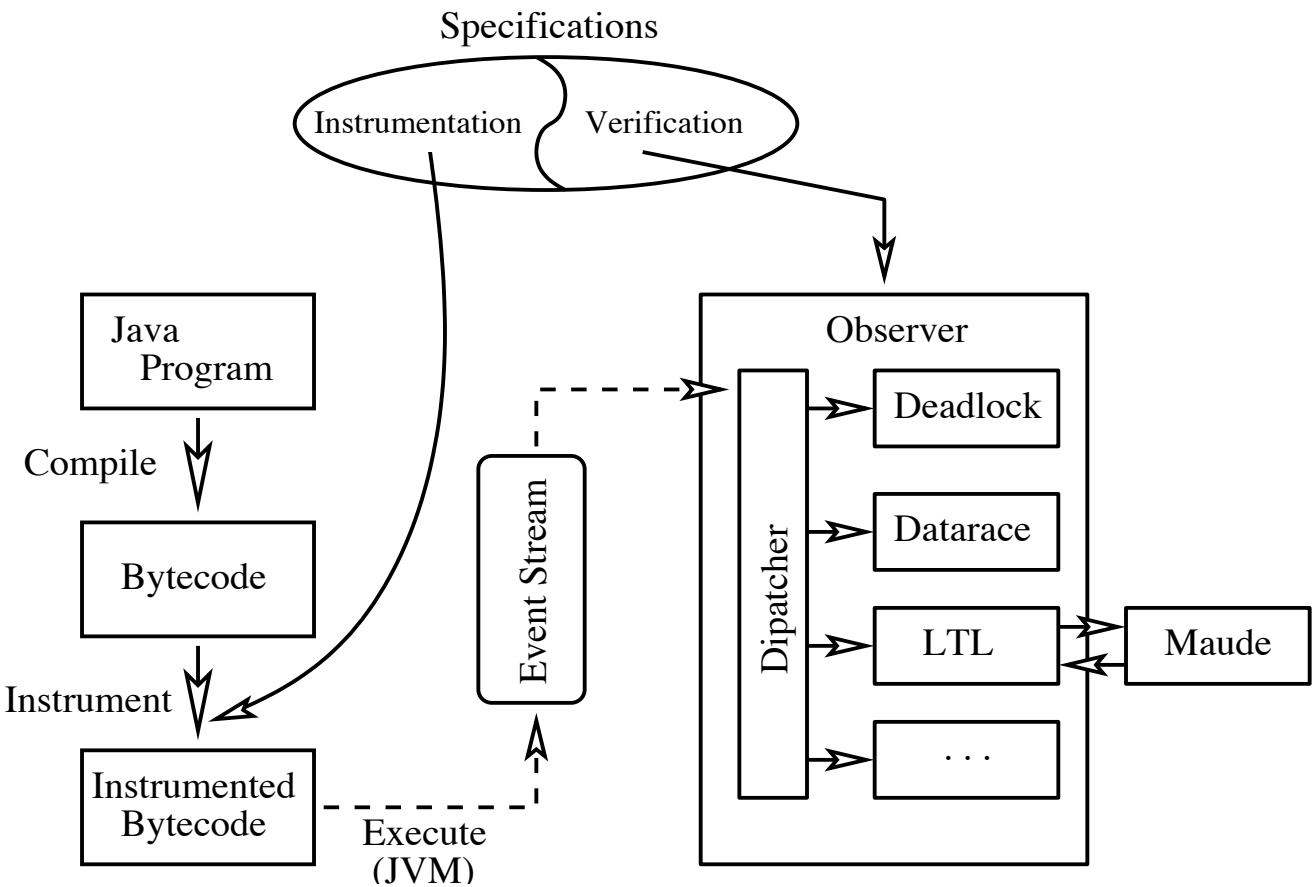
The example monitors here are deterministic.

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 mostly 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.

A real-life framework: Java Path Explorer

A runtime verification tool for Java (from NASA Ames)[3][4].

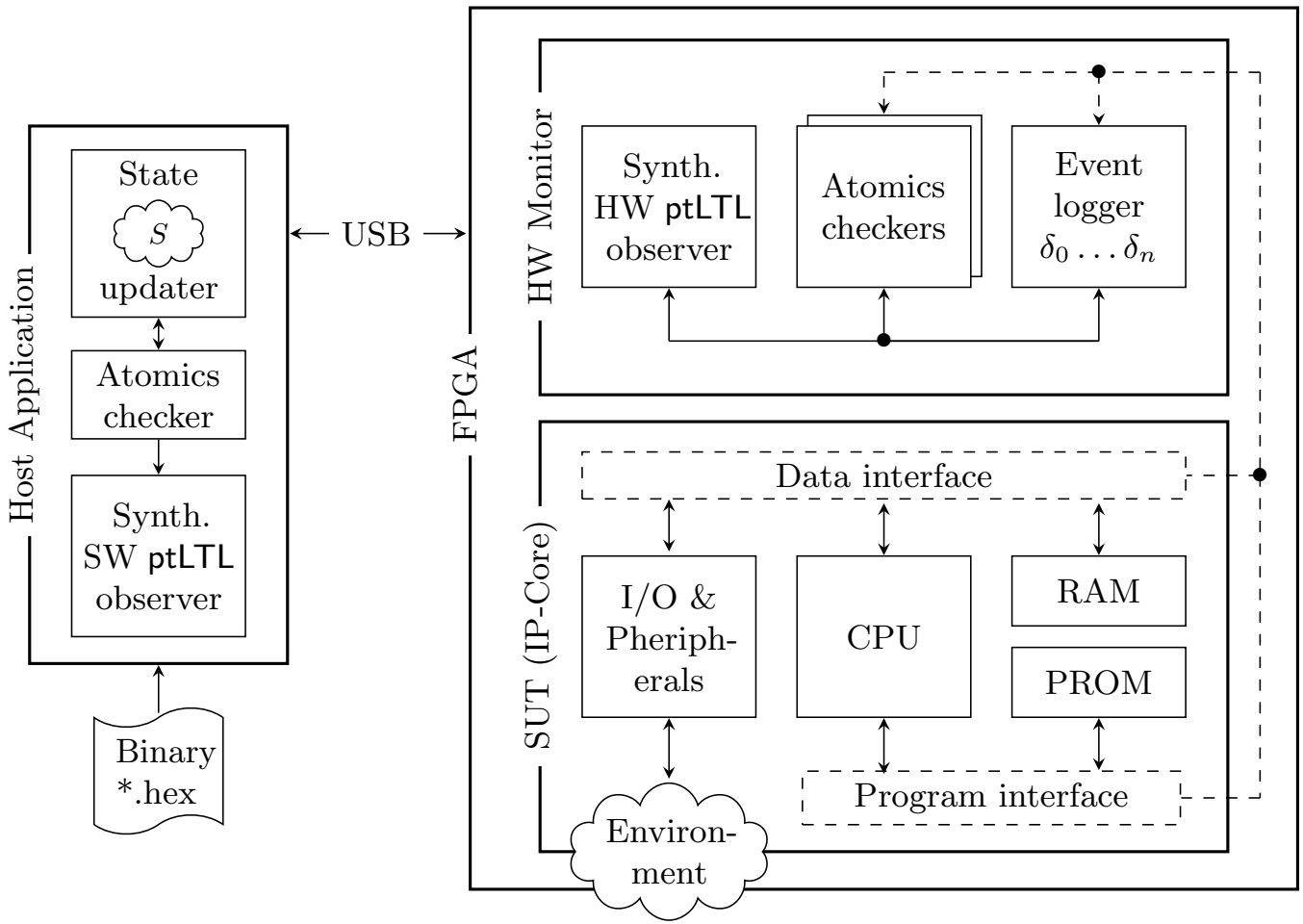


[3] An Overview of the Runtime Verification Tool Java PathExplorer. Klaus Havelund, Grigore Rosu. Formal Methods in System Design, Vol. 24, Issue 2, pp 189-215, 2004.

[4] Maude is an LTL model checker.

A real-life framework: Microcontroller monitoring

A runtime verification tool for binary code [5].



[5] Past Time LTL Runtime Verification for Microcontroller Binary Code. T. Reinbacher et al. Formal methods for industrial critical systems, pp 37-51, 2011. Best paper award.