Reactive Systems: Safety, Liveness, and Fairness

Timo Latvala

March 7, 2004

Safety

Safety properties are a very useful subclass of specifications. Basically safety properties are those properties, where a violation of the property can always be detected after only a finite execution of the system.

Thus model checking of safety properties requires only reasoning about finite executions (basically the history of the system execution so far). This makes algorithmics for model checking easier, as emptiness checking reduces basically to the reachability of a final state.

Violations of safety properties can also be detected with traditional testing and simulation methods.

Violations of safety properties can also often be monitored during implementation runtime. This is often called “runtime” verification.

Bad Prefixes

Let's now formalize this notion of safety. Consider a language L consisting a set of of infinite words over alphabet Σ, i.e., $L \subseteq \Sigma^\omega$.

A finite word $x \in \Sigma^*$ is a *bad prefix* for L iff for all infinite words $y \in \Sigma^\omega$ it holds that the concatenation $x \cdot y$ is *not* in L. In other words, no matter how x is extended, we will always get words not in L.

You can think of the bad prefix x as a finite counterexample showing that the safety property is violated.

Safety Languages

A language $L \subseteq \Sigma^\omega$ is a safety language (also called a safety property) iff every infinite word *not* in L has a finite bad prefix.

I.e., L is a safety language iff $\forall w \in \Sigma^\omega \setminus L$ there is $x \in \Sigma^*$ such that: (i) $w = x \cdot z$ for some $z \in \Sigma^\omega$, and (ii) for all $y \in \Sigma^\omega$ it holds that $x \cdot y \notin L$.

Safety and LTL

There is a subset of LTL properties, which is called “syntactic LTL safety formulas”. Basically this subset contains all the formulas built using the following syntax:

Given the set AP, an syntactic safety LTL formula is:

- **true**, **false**, p for $p \in AP$, or $\neg p$ for $p \in AP$,
- $f_1 \vee f_2$, $f_1 \wedge f_2$, $X f_1$, $\Box f_1$, or $f_1 R f_2$, where f_1 and f_2 are syntactic safety LTL formulas

Note that negation can only be applied to atomic propositions, thus until properties can not be expressed in the subset.

Automata for Syntactic LTL Safety Formulas

For syntactic safety formulas we have the following theorem:

Theorem 1 *Given a syntactic safety LTL formula f, there is a nondeterministic finite automaton $\mathcal{A}_{\neg f}$ (on finite words), which has at most $2^{O(|f|)}$ states, accepts only bad prefixes for $\mathcal{L}(f)$, and for each word $w \in \mathcal{L}(\neg f)$, the automaton accepts some finite word x, such that $w = x \cdot z$ for some infinite word z.*

The procedure to obtain that automaton is similar in spirit to LTL to Büchi conversion, but the details differ!

If you need this automaton, take a look at the “scheck” tool by T. Latvala (`http://www.tcs.hut.fi/~timo/scheck`).

Automata for General LTL Safety Formulas

Note that there are a lot of other LTL formulas which are safety, but are not in the syntactic safety subset. For example any formula which is equivalent to $false$ is a safety formula! Example: $\Diamond\Box p \wedge \Diamond\Box\neg p$.

If we get outside syntactic safety subset, things get a quite bit more messy:

Theorem 2 *Given a safety LTL formula f, there is a deterministic finite automaton $\mathcal{A}_{\neg f}$ (on finite words), which has at most $2^{2^{O(|f|)}}$ states, accepts only bad prefixes for $\mathcal{L}(f)$, and for each word $w \in \mathcal{L}(\neg f)$, the automaton accepts the shortest finite word x, such that $w = x \cdot z$ for some infinite word z.*

There is also a $2^{2^{\Omega(\sqrt{|f|})}}$ lower bound. Because of the blow-up, handling non-syntactic safety formulas can actually be made more efficiently by using Büchi automata instead!

Liveness

There are several different definitions of liveness in the literature. We define liveness as follows:

A language $L \subseteq \Sigma^\omega$ is a liveness language (also called a liveness property) iff L is *not* a safety language.

To detect violations of liveness properties thus need to consider (at least some) infinite executions of the system.

Fairness

Sometimes we want to use fairness assumptions on the environment out system works in.

For example, we might want to assume that a scheduler never ignores some process forever. It could be that a system can only guarantee progress if such a scheduler is present. However, optimally we would like our program to work even if the scheduler is very unfair. To model “the worst possible scheduler”, we might add fairness conditions implying that each process is scheduled infinitely often.

Another example of fairness assumptions is that of a lossy message channel, which will for each message “m_i” guarantee the following: If message “m_i” is sent infinitely often, then the same message “m_i” is also received infinitely often.

Note that fairness assumptions are only needed to prove liveness properties! Any safety property can be verified without assuming fairness.

Model Checking Under Fairness

In model checking under fairness, some fairness assumption is assumed from a system, such as that the used scheduler will schedule all processes infinitely often. This can often be captured by an LTL formula of the form

$$(\mathit{fairness}) \rightarrow (\mathit{property}).$$

One should be careful when specifying the formula for fairness, because it is easy to make a mistake and to specify a fairness assumption, which is equivalent to **false**.

Two very commonly used forms of fairness are weak fairness and strong fairness.

Weak fairness can be captured by using the LTL formula

$$\bigwedge_{1 \leq i \leq n} (\Diamond \Box p_i \rightarrow \Box \Diamond q_i),$$

which is actually equivalent to

$$\bigwedge_{1 \leq i \leq n} (\Box \Diamond (\neg p_1 \vee q_i)).$$

This formula can be translated into a one state (generalized) Büchi automaton, provided that the automaton class used has acceptance sets on arcs instead being on the states (as in the standard definition used in this course).

Most LTL to Büchi translators will (unfortunately) generate an exponential Büchi automaton when confronted with a weak fairness formula.

Thus it is advisable to see whether the LTL model checker you use handles weak fairness constraints in an efficient manner. Sometimes using an a better LTL to Büchi conversion tool can significantly improve the performance of model checking LTL under weak fairness.

The strong fairness is characterized by the LTL formula

$$\bigwedge_{1 \leq i \leq n} (\Box\Diamond p_i \rightarrow \Box\Diamond q_i).$$

Unfortunately, this cannot be translated into a one state Büchi automaton, and the exponential blowup is unavoidable.

It can, however, be translated into one state automaton of a class called a *Streett* automaton (again provided that the acceptance conditions are on edges).

The acceptance component of a (state acceptance set based) Streett automaton is $\Omega = \{(L_1, U_1), (L_2, U_2), \ldots, (L_n, U_n)\}$. A run r of the Streett automaton $\mathcal{A}$ is accepting iff

$$\bigwedge_{i=1}^{n} (inf(r) \cap L_i = \emptyset \vee inf(r) \cap U_i \neq \emptyset).$$

It is easy to see that the acceptance component is basically a strong fairness formula.

Generalized Büchi automata can be emulated by Streett automata by setting $L_i = S$, and $U_i = F_i$ for all i. However, the other direction involves an exponential blowup.

Upgrading your LTL to Büchi translator to a decent one might help some when dealing with model checking under strong fairness, but the resulting Büchi automata will always be exponential in the number of fairness constraints.

The emptiness checking algorithms for Streett automata are more complex than those of Büchi automata. They are, however, still polynomial.

Thus if you need to model check under many strong fairness constraints, using a model checker employing Streett automata is advisable.

(Use e.g., the Petri net model checker of the Maria tool, due to T. Latvala and K. Heljanko.)