T-79.186 Reactive Systems

Spring 2003, Lecture 10

Keijo Heljanko

March 4, 2003

11 On-the-fly Model Checking

An model checking algorithm, which can terminate “early” with a yes/no answer to a model checking question, is called an on-the-fly model checker.

For example, in LTL model checking we can find the following phases:

1. Generation of a Büchi automaton $\mathcal{A}_{\neg f}$
2. Calculation of the product automaton $\mathcal{P} = \mathcal{A}_M \otimes \mathcal{A}_{\neg f}$, where often $\mathcal{A}_M = \mathcal{A}_1 \otimes \mathcal{A}_2 \otimes \cdots \otimes \mathcal{A}_n$. Thus, in fact, in practice the product is simultaneously composed out of $n+1$ automata as follows: $\mathcal{P} = \mathcal{A}_1 \otimes \mathcal{A}_2 \otimes \cdots \otimes \mathcal{A}_n \otimes \mathcal{A}_{\neg f}$.
3. Checking the emptiness of the product automaton $\mathcal{P}$, i.e., whether $\mathcal{L}(\mathcal{P}) \neq \emptyset$.

An “on-the-fly LTL model checker” in practice combines the phases 2 and 3 above into one subroutine.

A popular candidate for the phase 3 is to use the nested DFS algorithm, with initial states and the successor relation provided by a subroutine implementing the phase 2 automata synchronization.

In practice phase 1 is run off-line, because doing so provides many optimization possibilities which are hard to do on-the-fly.

An on-the-fly CTL model checker can be done in a similar fashion. This, however, requires the use of a CTL model checking algorithm, which does not require the predecessor relation. This slightly complicates algorithm design, but suitable algorithms do exist. (See again the Master's Thesis by K. Heljanko for an overview.)

For an on-the-fly CTL^* model checker one needs a (global) on-the-fly LTL model checker. Implementing it based on the nested DFS can be non-optimal, and possibly an algorithm based on Tarjan's MSCC algorithm should be used instead.

12 Abstraction

We have had talks about abstraction already in this course, so here is just a slightly different view of things.

We will introduce Kripke structure equivalences and preorders.

The notions will then be used to describe what kinds of manipulations will preserve formulas of the logics LTL, CTL, and CTL^*.

12.1 Bisimulation

Bisimulation is the equivalence which is characterized by the logic CTL^*.

Definition 12.1 Let $M = (S, s^0, R, L)$ and $M' = (S', {s^0}', R', L')$ be Kripke structures with the same set of atomic propositions AP.

A relation $B \subseteq S \times S'$ is a **bisimulation relation** iff for all $s \in S, s' \in S'$, if $B(s, s')$ then the following conditions hold:

- $L(s) = L'(s')$
- For every $s_1 \in S$ such that $R(s, s_1)$ there is ${s_1}' \in S'$ such that $R'(s', {s_1}')$ and $B(s_1, {s_1}')$
- For every ${s_1}' \in S'$ such that $R(s', {s_1}')$ there is $s_1 \in S$ such that $R(s, s_1)$ and $B(s_1, {s_1}')$

Two Kripke structures M and M' are bisimulation equivalent (denoted $M \equiv M'$) iff there exists a bisimulation B, such that $B(s^0, {s^0}')$.

The following theorem states that bisimulation preserves CTL^*:

Theorem 12.2 If $M \equiv M'$ then for every CTL^* formula f, it holds that $M \models f$ iff $M' \models f$.

Thus, if one can prove that some manipulation of a model results in a (hopefully much smaller) Kripke structure M', which is still bisimulation equivalent to the original Kripke structure M, we can do the following.

Model check any CTL^* formula f on M' instead of M. (To save memory and space.)

Unfortunately, most of the manipulations on model level do not preserve bisimulation, as bisimulation requires that the branching structure of the program is preserved.

Bisimulation is the best we can do, as given by the following theorem.

Theorem 12.3 If $M \not\equiv M'$ then there exist a CTL formula f, such that $((M \models f \wedge M' \not\models f) \vee (M \not\models f \wedge M' \models f))$.

Thus, in fact, by preserving CTL one also preserves all of CTL^*.

If one considers the logic CTL^*-X, there also exists an equivalence characterizing it, usually called **stuttering bisimulation**. This equivalence is used by partial order reductions preserving all of CTL^*-X.

12.2 Simulation

A much more useful notion to be used in context of abstraction is that of a simulation.

Definition 12.4 Let $M = (S, s^0, R, L)$ and $M' = (S', {s^0}', R', L')$ be Kripke structures with $AP \subseteq AP'$.

A relation $H \subseteq S \times S'$ is a **simulation relation** iff for all $s \in S, s' \in S'$, if $H(s, s')$ then the following conditions hold:

- $L(s) \cap AP' = L'(s')$
- For every $s_1 \in S$ such that $R(s, s_1)$ there is ${s_1}' \in S'$ such that $R'(s', {s_1}')$ and $H(s_1, {s_1}')$

We say that M' simulates M (denoted by $M \preceq M'$), if there exists a simulation relation H, such that $H(s^0, {s^0}')$.

It is easy to show that $\preceq$ is a preorder (reflexive and transitive relation).

The logic $ACTL^*$ consist of those formulas of CTL^*, which in negation normal form do not contain the existential E path quantifier. Most notably all LTL formulas are also $ACTL^*$ formulas (once you add the implicit A in front of an LTL formula).

We have the following result.

Theorem 12.5 Suppose $M \preceq M'$. Then for every $ACTL^*$ formula f with atomic propositions from AP' it holds that $M' \models f$ implies $M \models f$.

Thus, if one comes up with an abstraction method, one needs to prove that it preserves simulation, and after that all universally quantified CTL^* formulas can be checked on the reduced structure M'.

Now any abstraction method, for which it can be proved that the Kripke structure M' of the abstracted system simulates the Kripke structure M of the original system is sound for the logics: $ACTL^*$, LTL, and safety subset(s) of LTL.

One should, however, be careful when deadlocks are present in the original system.

The "easiest" way is to remove deadlocks by adding a transition "$dummy$", which is enabled iff none of the transitions of the original system is, and which does not change the state of the system when it fires. When doing abstraction, also the "$dummy$" transition is abstracted, and therefore in the abstracted system the abstract version of "$dummy$", call it "$dummy'$" will always be enabled in states corresponding to deadlocks.

The second option is to prove that the original system is deadlock free (e.g., by construction).

The third option is to only verify safety properties after abstraction.