T-79.186 Reactive Systems

Spring 2003, Lecture 9

Keijo Heljanko

February 28, 2003

8.4 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

$$(fairness) \rightarrow (property).$$

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 $\mathbf{false}$.

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

Weak fairness can be captured by using the LTL formula

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

which is actually equivalent to

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

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 model checker you use handles weak fairness constraints in an efficient manner.

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

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

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.

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 model checker of the Maria tool, due to T. Latvala and K. Heljanko.)

9 Model Checking CTL

There is a straightforward model checker for CTL, whose running time is linear in both the size of the Kripke structure and the size of the formula.

We will now present a CTL model checking algorithm due to Emerson, Clarke, and Sistla.

In our presentation we use AU and EU as single subformulas. Also, we will only present AX and $\wedge$, as EX and $\vee$ can be obtained from the using negation.

For convenience we assume that the subformulas are numbered in an order, where $\mathit{left}(f)$ and $\mathit{right}(f)$ have smaller indexes than f.

The algorithm uses one bit-array of size $|S|$ called "label" for each subformula to store the truth values of different subformulas.

One bit-array of size $|S|$ called "marked" is also used for internal bookkeeping.

This algorithm uses both successor and predecessor lists of a state in the Kripke structure.

For an on-the-fly implementation another (more complex) algorithm is needed (see e.g., Master's Thesis by K. Heljanko for an overview of on-the-fly CTL model checking algorithms).

Algorithm 1 Main loop of a CTL model checker.

```
procedure global_model_checker(f)
    for i := 1 to length(f) do
        foreach s ∈ S do
            reset_label(s, i); // Init formula i to false
        od
        label_graph(i); // Evaluate the formula f_i in all states
    od
end procedure
```

Algorithm 2 Choose processing subroutine based on formula type

```
procedure label_graph(i)
    ftype := formula_type(f_i);
    if ftype = atomic_proposition then
        atomic(i);
    elsif ftype = NOT then
        negation(i);
    elsif ftype = AND then
        conjunction(i);
    elsif ftype = AX then
        ax(i);
    elsif ftype = AU then
        au(i);
    elsif ftype = EU then
        eu(i);
    endif
end procedure
```

Algorithm 3 Process atomic proposition

```
procedure atomic(i)
    foreach s ∈ S do
        if evaluate_proposition(s, i) then
            add_label(s,i);
        endif
    od
end procedure
```

Algorithm 4 Process negation

```
procedure negation(i)
    foreach s ∈ S do
        if ¬labeled(s, left(i)) then
            add_label(s,i);
        endif
    od
end procedure
```

Algorithm 5 Process conjunction

```
procedure conjunction(i)
    foreach s ∈ S do
        if labeled(s, left(i)) ∧ labeled(s, right(i)) then
            add_label(s, i);
        endif
    od
end procedure
```

Algorithm 6 Process universal next-state formula

```
procedure ax(i)
    foreach s ∈ S do
        add_label(s, i);
        foreach t ∈ successors(s) do
            if ¬labeled(t, left(i)) then
                reset_label(s, i);
                break;
            endif
        od
    od
end procedure
```

Algorithm 7 Process universal until formula

```
procedure au(i)
    foreach s ∈ S do
        reset_marked(s);
    od
    foreach s ∈ S do
        if ¬marked(s) then
            check_au(i, s);
        endif
    od
end procedure
```

Algorithm 8 Process universal until formula for state s

```
procedure check_au(i, s)
    if marked(s) then
        if labeled(s, i) then
            return true;
        else
            return false;
        endif
    endif
```

```
    set_marked(s);
    if labeled(s, right(i)) then
        add_label(s, i);
        return true;
    elsif ¬labeled(s, left(i)) then
        return false;
    endif
    foreach t ∈ successors(s) do
        if ¬check_au(i, t) then
            return false;
        endif
    od
    add_label(s, i);
    return true;
end procedure
```

Algorithm 9 Process existential until formula

```
procedure eu(i)
    foreach s ∈ S do
        reset_marked(s);
    od
    foreach s ∈ S do
        if ¬marked(s) then
            check_eu(i, s);
        endif
    od
end procedure
```

Algorithm 10 Process existential until formula for state s

```
procedure check_eu(i, s)
    if labeled(s, right(i)) then
        add_label(s, i);
        label_predecessors(i, s);
    endif
end procedure
```

Algorithm 11 Propagate label change to predecessor states

```
procedure label_predecessors(i, s)
    set_marked(s);
    foreach t ∈ predecessors(s) do // Note the use of predecessor relation!
        if ¬marked(t) ∧ labeled(t, left(i)) then
            add_label(t, i);
            label_predecessors(i, t);
        endif
    od
end procedure
```

10 Model checking CTL^*

Model checking CTL^* is quite straightforward once we have a global model checker for LTL. (An algorithm which evaluates the LTL formula in all states of the system.)

Assume we have an (existential) LTL model checker, which (in CTL^* notation) returns the set of states $\{s \in S \mid M, s \models Ef_1\}$, where f_1 is an LTL formula.

We call this algorithm "$ECheckLTL()$".

We will now show that model checking CTL^* can be made with an algorithm of essentially the same complexity as the complexity of "$ECheckLTL()$" by using the following procedure.

The recursive evaluation procedure "$CheckCTL^*(f)$" goes as follows:

1. Convert the CTL^* formula f into negation normal form. (Push negations in).
2. If f is of the form Ef_1, where f_1 is and LTL formula, return $ECheckLTL(f_1)$.
3. If f is of the form Af_1, return $(S \setminus CheckCTL^*(E\neg f_1))$.
4. Let $g_1, g_2, \ldots, g_n$ be the maximal subformulas of f, which are not LTL formulas. For each g_i, create a new atomic proposition h_i, and calculate the valuation of it by calling $CheckCTL^*(g_i)$. Furthermore, replace each subformula g_i in the formula f by the corresponding atomic proposition h_i.
5. return $ECheckLTL(f)$.
   (After the step 4. above, f is guaranteed to be an LTL formula.)

Now $M, s^0 \models f$ iff $s^0 \in CheckCTL^*(f)$.

To implement the global LTL model checking procedure "$ECheckLTL(f_1)$", one can for example call a standard (local) LTL model checking procedure with the formula $\neg f_1$ and negate the result. Calling this procedure $|S|$ times, each time with a new initial state, will calculate the required set of states.

However, doing so will not be very efficient, as a lot of redundant is done across the different calls. (The global model checking algorithm is now quadratic in the number of states in the Kripke structure, instead of being linear.)

Using a modified version of the nested depth first search will get rid of most of the overhead in practice, but the quadratic worst case behavior remains.

By using an MSCC based emptiness checking algorithm (e.g., modified Tarjan's MSCC algorithm) this quadratic overhead can be eliminated.

Note, however, that using CTL^* instead of LTL has also disadvantages. For example, partial order reduction methods for CTL^* are less effective than for LTL. Also, the use of abstraction methods is more cumbersome for CTL^*.

Thus even though model checking as such is not harder for CTL^*, practical model checking use might still prefer LTL over it.