T-79.186 Reactive Systems

Spring 2003, Lecture 6

Keijo Heljanko

February 25, 2003

8.2 Algorithms for Büchi Automata

The following algorithm can be used to check a Büchi automaton for emptiness.

It uses a hash table to check whether a state s has already been visited by the algorithm. A new state can be stored into this table using subroutine "hash(s)". For efficiency each hash table entry contains two bits of additional information, both initialized to zero value.

To manipulate these bits, there are the following subroutines. The subroutine "addstack1(s)" turns the first bit to one, the subroutine "removestack1(s)" clears the first bit, and the subroutine "instack1(s)" returns "True" iff the first bit is set.

The subroutine "flag(s)" turns the second bit to one, and the subroutine "flagged(s)" returns "True" iff the second bit is set.

Algorithm 1 The top-level nested DFS algorithm

procedure emptiness
    for all $s \in S^0$ **do**
        dfs1(s);
    terminate(False); // Automaton is empty
end procedure

Algorithm 2 The dfs1 subroutine

procedure dfs1(s)
    local s';
    hash(s);
    addstack1(s);
    for all successors s' of s **do** // ($s' \in \rho(s, a)$ for some $a \in \Sigma$)
        if s' is not in the hash table **then** dfs1(s');
    if s is an accepting state **then** dfs2(s); // ($s \in F$)
    removestack1(s);
end procedure

Algorithm 3 The dfs2 subroutine

procedure dfs2(s)
    local s';
    flag(s);
    for all successors s' of s **do** // ($s' \in \rho(s, a)$ for some $a \in \Sigma$)
        if instack1(s') **then terminate**(True); // Accepting run through s' found!
        else if not flagged(s') **then** dfs2(s');
        end if;
end procedure

Actually DFS search order is needed for correctness only in the subroutine "dfs1(s)". (Using DFS there is vital for correctness!)

The subroutine "dfs2(s)" can actually be implemented using any search order (for example BFS). However, doing so requires a data structure for storing the value of "flag" different form the one described in the previous slides.

The above emptiness checking algorithm "nested depth first search" is what is implemented in the LTL model checker SPIN.

8.3 Translating LTL into Büchi Automata

There are several algorithms for translating LTL formulas into Büchi automata. In this course we will go through a variant due to Gerth, Peled, Vardi, and Wolper.

Given an LTL formula f, it will generate a Büchi automaton $\mathcal{A}_f$ of with at most $2^{\mathcal{O}(|f|)}$ states.

The automaton $\mathcal{A}_f$ will accept the language $\{w \in (\Sigma)^\omega \mid w \models f\}$, where $\Sigma = 2^{AP}$.

Recall that if we want to model check an LTL property h, we should actually create an automaton for $f = \neg h$.

Before we proceed any further, we want to put the formula f into negation normal form (also called positive normal form), where all negations appear appear only in front of atomic propositions.

This can be done with previously presented DeMorgan rules for temporal logic operators. Note that putting the formula into positive normal form does not involve a blow-up. (The length of the formula at most doubles.)

We will keep f as the name of the formula after this procedure.

We will now define the set of formulas $sub(f)$ to be the smallest set of LTL formulas satisfying all of the following conditions:

- Boolean constants $\mathbf{true}, \mathbf{false}$, and the top-level formula f belong to $sub(f)$,
- if $f_1 \vee f_2 \in sub(f)$, then $\{f_1, f_2\} \subseteq sub(f)$
- if $f_1 \wedge f_2 \in sub(f)$, then $\{f_1, f_2\} \subseteq sub(f)$
- if $X\, f_1 \in sub(f)$, then $\{f_1\} \subseteq sub(f)$
- if $f_1\, U\, f_2 \in sub(f)$, then $\{f_1, f_2\} \subseteq sub(f)$
- if $f_1\, R\, f_2 \in sub(f)$, then $\{f_1, f_2\} \subseteq sub(f)$

It is easy to show that $|sub(f)| = \mathcal{O}(|f|)$.

To ease implementation, the formulas of $sub(f)$ can be numbered, and thus any subset of $sub(f)$ can be represented with a bit-array of length $|sub(f)|$, and thus there are at most $2^{\mathcal{O}(|f|)}$ such subsets.

The basic idea of the translation is based on the following properties of the semantics of LTL, where we can choose which way to prove a particular property:

- To prove that $w \models f_1 \vee f_2$ it suffices to either prove that
  a) $w \models f_1$, or
  b) $w \models f_2$.
- To prove that $w \models f_1 \, U \, f_2$ it suffices to either prove that
  a) $w \models f_2$, or
  b) $w \models f_1$ and $w \models X(f_1 \, U \, f_2)$.
- To prove that $w \models f_1 \, R \, f_2$ it suffices to either prove that
  a) $w \models f_1$ and $w \models f_2$, or
  b) $w \models f_2$ and $w \models X(f_1 \, R \, f_2)$.

The only restriction being, that when proving $f_1 \, U \, f_2$ the case b) can only be used infinitely often iff the case a) is used infinitely often (we will use Büchi acceptance sets to handle that).

The algorithm will manipulate a data structure called **node** during its run.

A node is a structure with the following fields:

- ID: A unique identifier of a node (a number),
- $Incoming$: A list of node IDs,
- $Old \subseteq sub(f)$,
- $New \subseteq sub(f)$, and
- $Next \subseteq sub(f)$.

The nodes will form a graph, where the arcs of the graph are stored in the $Incoming$ list of the end node of the arc for easier manipulation.

The initial node is marked by having a special node ID called $init$ in its $Incoming$ list.

All nodes are stored in a set (use e.g., a hash table for implementation) called $nodes$.

To implement the algorithm, the following functions are defined.

- $Neg(\mathbf{true}) = \mathbf{false}$,
- $Neg(\mathbf{false}) = \mathbf{true}$,
- $Neg(p) = \neg p$ for $p \in AP$, and
- $Neg(\neg p) = p$ for $p \in AP$.

The functions $New1(f)$, $Next1(f)$, and $New2(f)$ are tabulated below. They match the recursive definitions for disjunction, until, and release:

f	$New1(f)$	$Next1(f)$	$New2(f)$
$f_1 \vee f_2$	$\{f_2\}$	$\emptyset$	$\{f_1\}$
$f_1 \, U \, f_2$	$\{f_1\}$	$\{f_1 \, U \, f_2\}$	$\{f_2\}$
$f_1 \, R \, f_2$	$\{f_2\}$	$\{f_1 \, R \, f_2\}$	$\{f_1, f_2\}$

We are now ready to present the translation algorithm. (The presented version does not handle $X \, f_1$, but it can be easily added.)