Reactive Systems: Automata on Infinite Words, part II

Timo Latvala

February 25, 2004

Kripke Product

We'll now show a small trick, using which $\mathcal{A}_M \times \mathcal{A}_{\neg f}$ can be replaced by a slightly smaller automaton, which we call a Kripke product, and denote by $\mathcal{A}_M \otimes \mathcal{A}_{\neg f}$.

In the special case $F_1 = S_1$ we can actually use a simpler product construction, we denote it by $\mathcal{A}_1 \otimes \mathcal{A}_2$:

Let $\mathcal{A}_1 = (\Sigma, S_1, S_1^0, \Delta_1, F_1)$ and $\mathcal{A}_2 = (\Sigma, S_2, S_2^0, \Delta_2, F_2)$ be two Büchi automata, such that for automaton $\mathcal{A}_1$ it holds that $F_1 = S_1$. (Note that this is the case when $\mathcal{A}_1$ is generated from a Kripke structure.)

Definition 1 *We define the* Kripke product *automaton to be* $\mathcal{A} = (\Sigma, S, S^0, \Delta, F)$*, where:*

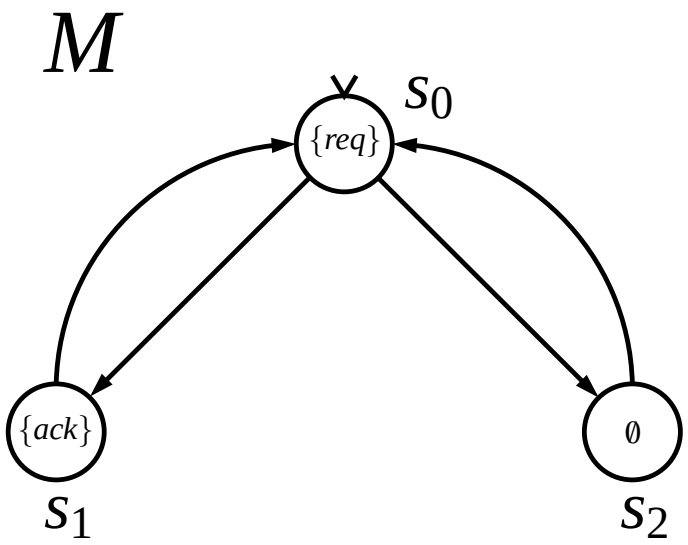
- $S = S_1 \times S_2$,
- $S^0 = S_1^0 \times S_2^0$,
- *for all* $s, s' \in S_1, t, t' \in S_2, a \in \Sigma$*:*
  $((s,t),a,(s',t')) \in \Delta$ *iff* $(s,a,s') \in \Delta_1$ *and* $(t,a,t') \in \Delta_2$*; and*
- $F = S_1 \times F_2$.

Now for the Kripke product Büchi automaton $\mathcal{A}$ (also denoted by $\mathcal{A}_1 \otimes \mathcal{A}_2$) it holds that $\mathcal{L}(\mathcal{A}) = \mathcal{L}(\mathcal{A}_1) \cap \mathcal{L}(\mathcal{A}_2)$. (Side note: The above definition happens to be equivalent to the FSA $\mathcal{A} = \mathcal{A}_1 \cap \mathcal{A}_2$ operation because $S_1 = F_1$!)

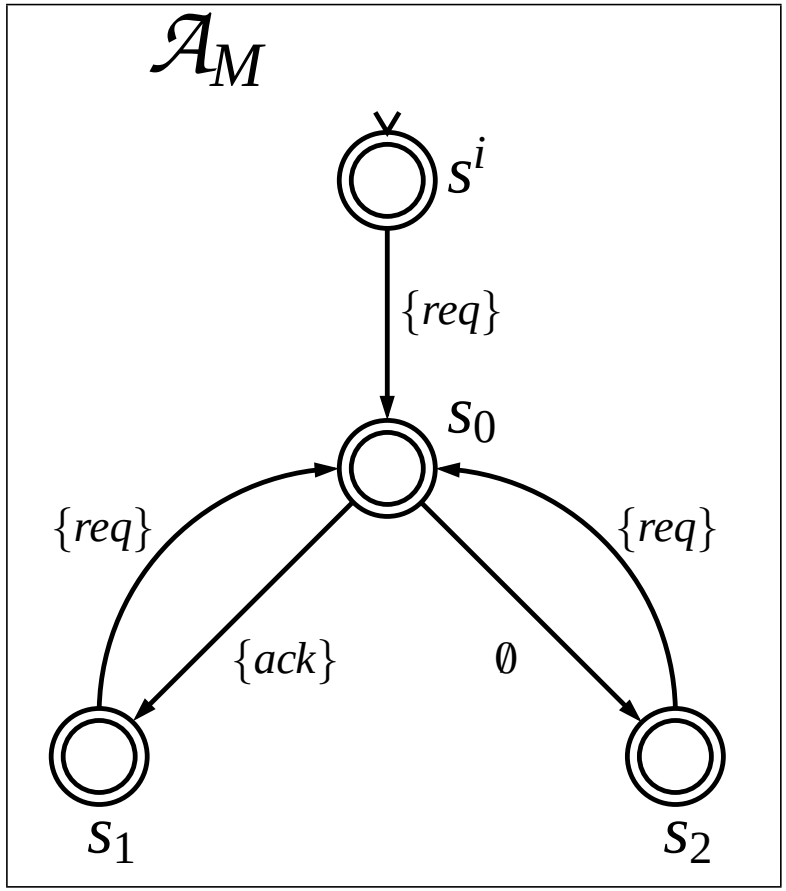
Now if $\mathcal{A}_1$ is a Kripke structure automaton $\mathcal{A}_M$, it fulfills the property above, and thus this *Kripke product* construction can be used instead. (It has half as many states.)

Example: LTL Model Checking

Assume we want to check whether $M \models f$, where $f = \Box(req \Rightarrow (\Diamond ack))$ for the Kripke structure M below. This can be solved by checking whether for the Kripke product automaton $\mathcal{P} = \mathcal{A}_M \otimes \mathcal{A}_{\neg f}$ it holds that $\mathcal{L}(\mathcal{P}) = \emptyset$. If so, then $M \models f$, otherwise $M \not\models f$.

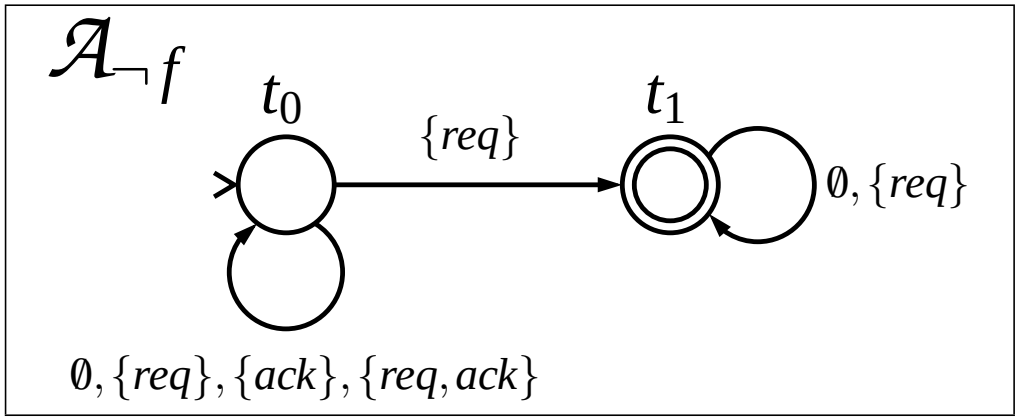


Büchi Automaton $\mathcal{A}_M$

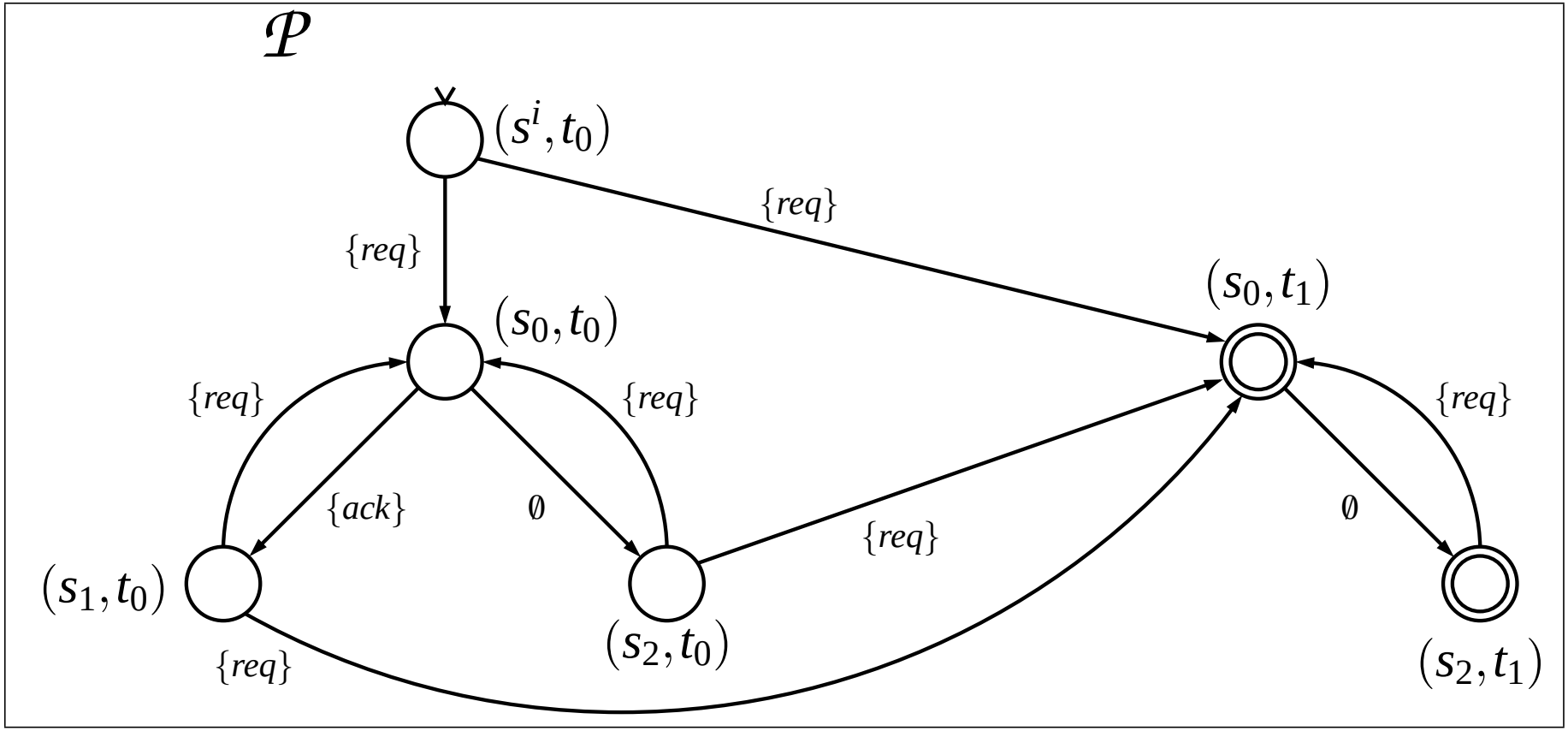


Büchi Automaton $\mathcal{A}_{\neg f}$

Now it is easy to see that $\neg f = \Diamond(req \wedge (\Box \neg ack))$.



Büchi Automaton $\mathcal{P}$



Büchi Automaton $\mathcal{P}$ is Non-empty!

It is easy to see that the Büchi automaton $\mathcal{P}$ has accepting runs. An example is the run $r = (s^i, t_0), (s_0, t_1), (s_2, t_1), (s_0, t_1), (s_2, t_1), \ldots$. Now $inf(r) = \{(s_0, t_1), (s_2, t_1)\}$, and thus the run r is accepting.

The word accepted by r is: $w = \{req\}, \emptyset, \{req\}, \emptyset, \{req\}, \emptyset, \ldots$.

The path π the run r corresponds to can be obtained from r by projecting r on the first component, and dropping the special state s^i from the beginning, i.e., $\pi = s_0, s_2, s_0, s_2, \ldots$.

We also get that $\pi \models \neg f$, and finally that $M \not\models f$.

Emptiness Checking for Büchi Automata

The following “nested depth-first search” 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 (only) 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 *state* s'*;*

    hash(s);

    addstack1(s);

    for all *successors* s' *of* s **do** *//* $((s, a, s') \in \Delta$ *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 state s';
    flag(s);
    for all successors s' of s **do** // $((s, a, s') \in \Delta$ 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 from 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.

Maximal Strongly Connected Components

We define a strongly connected component C of a directed graph to be a set of nodes $C \subseteq S$, in which for all pairs of distinct states $s, s' \in C$ it holds that: s' can be reached from s and s can be reached from s'.

A strongly connected component C is called maximal, if no strongly connected component $C' \subseteq S$ exists, such that $C \subset C'$.

A maximal strongly connected component is called non-trivial iff: (i) $|C| > 1$, or (ii) there exists $s \in C$ such that there is an edge in the graph from s back to s.

Emptiness Checking with MSCCs

Another way of checking the non-emptiness of $\mathcal{L}(\mathcal{A})$ is to compute the maximal strongly connected components (MSCCs) of the Büchi automaton, and check whether some non-trivial maximal strongly connected component C reachable from some initial state $s \in S^0$ contains an accepting state ($C \cap F \neq \emptyset$). If so, the language is non-empty. Otherwise, the language is empty.

Also this emptiness checking approach can be implemented with a linear time algorithm, e.g. by using the Tarjan's algorithm for computing the MSCCs. (Compute the reachable MSCCs and check whether any non-trivial MSCC contains an accepting state.)

The definition of determinism for Büchi automata is identical to the FSA case.

Unlike finite state automata, Büchi automata are not expressively complete when deterministic. In other words, there are languages accepted by non-deterministic Büchi automata, which no deterministic Büchi automaton accepts. An example of such a language is $(a+b)^* b^\omega$. (A finite number of a symbols with finitely many occurrences of b symbols between any two a's followed by an infinite sequence of b symbols.)

Also note that LTL requires non-deterministic automata to be expressed, the language above is effectively the same as the requirement expressed by the LTL formula $\Diamond\Box b$.

The complementation procedure for Büchi automata is thus very different from finite state automata, as a normal determinization construction cannot be used. In fact, we have the following result:

Theorem 2 *Let $\mathcal{A}$ be any (non-deterministic) Büchi automaton with n states. Then in the worst case the smallest Büchi automaton $\mathcal{A}'$, such that $\mathcal{L}(\mathcal{A}') = \Sigma^\omega \setminus \mathcal{L}(\mathcal{A})$ will have $2^{\Omega(n \log n)}$ states.*

This blow-up is in practice much worse than the blow-up for finite state automata complementation. Note that $n! = O(2^{O(n \log n)})$. (For example, while $5! = 120$ and $2^5 = 32$, factorial grows much faster: $10! = 3628800$, while $2^{10} = 1024$.)

There are several different ways to complement Büchi automata matching the lower bound. However, we do not know of a publicly available implementation of those algorithms.

Thankfully in (basic) model checking complementation of Büchi automata is not needed.