T-79.186 Reactive Systems

Spring 2003, Lecture 2

Keijo Heljanko

January 22, 2003

5 Finite State Automata – Part II

The operations we have defined for finite state automata so far have resulted in automata whose size is polynomial in the sizes of input automata.

The most straightforward way of implementing complementation of a non-deterministic automaton is to first determinize it, and then complement the results.

Unfortunately determinization yields an exponential blow up. (A worst-case exponential blow-up is in fact unavoidable in complementing non-deterministic automata.)

Determinization of finite state automata can be done as follows:

Definition 5.1 Let $\mathcal{A}_1 = (\Sigma, S_1, S_1^0, \rho_1, F_1)$ be a non-deterministic automaton. We define a deterministic automaton $\mathcal{A} = (\Sigma, S, S^0, \rho, F)$, where

- $S = 2^{S_1}$, the set of all sets of states in S_1,
- $S^0 = \{S_1^0\}$, a single state having as label the set of initial states of $\mathcal{A}_1$,
- $\rho(s, a) = \{t \,|\, t \in \rho_1(s', a) \text{ for some } s' \in s \}$, and
- $F = \{s \in S \,|\, S \cap F_1 \neq \emptyset\}$, those states in S which contain at least one accepting state of $\mathcal{A}_1$.

The intuition behind the construction is that it combines all possible runs on given input word into one run over a larger state set. (The state set remembers all states in which the automaton can be in after reading the input so far.)

Note that $L(\mathcal{A}) = L(\mathcal{A}_1)$, and $\mathcal{A}$ is deterministic. If $\mathcal{A}_1$ has n states, the automaton $\mathcal{A}$ will contain 2^n states.

We have now shown that finite state automata are closed under all Boolean operations, as with $\cup$, $\cap$, and $\overline{\mathcal{A}}$ all other Boolean operations can be done.

All operations except for determinization created a polynomial size output in the size of the inputs.

Note, however, that even if $\mathcal{A}_1, \mathcal{A}_2, \mathcal{A}_3, \mathcal{A}_4$ have k states each, the automaton $\mathcal{A}'_4 = ((((\mathcal{A}_1 \cap \mathcal{A}_2) \cap \mathcal{A}_3) \cap \mathcal{A}_4)$ can have k^4 states, and thus in the general $\mathcal{A}'_i$ will have k^i states. (Note: The parenthesis above can and will often be dropped.)

Therefore even if if a single use of $\cap$ is polynomial, repeated applications often will result in a state explosion problem.

In fact, the use of $\cap$ as demonstrated above is the main way of composing system out of its components. The exact definition of the product operator $\times$ (used here to implement $\cap$) differs slightly in other state machine based formalisms, but they all suffer from a similar state explosion problem.

5.1 Checking Safety Properties with FSA

A safety property be informally described as a property stating that "nothing bad should happen". (We will come back to the formal definition later in the course.)

When checking safety properties, the behavior of a system can be described by a finite state automaton, call it $\mathcal{A}$.

Also in most cases the allowed behaviors of the system can be specified by another automaton, call it the specification automaton $\mathcal{S}$.

Assume that the specification specifies all legal behaviors of the system. In other words a system is incorrect if it has some behavior (accepts a word) that is not accepted by the specification. In other words a correct implementation has less behavior than the specification, or more formally $L(\mathcal{A}) \subseteq L(\mathcal{S})$.

Checking whether $L(\mathcal{A}) \subseteq L(\mathcal{S})$ holds is referred to as performing a language containment check.

By using simple automata theoretic constructions given above, we can now check whether the system meets its specification. Namely, we can create a product automaton $\mathcal{P} = \mathcal{A} \cap \overline{\mathcal{S}}$ and then check whether $L(\mathcal{P}) = \emptyset$.

In case the safety property does **not** hold, the automaton $\mathcal{P}$ has a counterexample run r_p which accepts a word w, such that $w \in L(\mathcal{A})$ but $w \notin L(\mathcal{S})$.

By projecting r_p on the states of $\mathcal{A}$ one can obtain a run of r_a of the system (a sequence of states of the system) which violates the specification $\mathcal{S}$.

6 Kripke Structures

Temporal logics are traditionally defined in terms of **Kripke structures**. A Kripke structure is basically a graph having the reachable states of the system as nodes and state transitions of the system as edges. It also contains a labelling of the states of the system with properties that hold in each state.

Definition 6.1 Let AP be a non-empty set of atomic propositions. A Kripke structure is a four tuple $M = (S, s^0, R, L)$, where

- S is a finite set of states,
- $s^0 \in S$ is an initial state,
- $R \subseteq S \times S$ is a transition relation, for which it holds that $\forall s \in S : \exists s' \in S : (s, s') \in R$, and
- $L : S \to 2^{AP}$ is labelling, a function which labels each state with the atomic propositions which hold in that state.

Kripke structures are the model used to give semantics (definition of when a formula holds) for the most widely used specification languages for reactive systems, namely **temporal logics**.

Kripke structures can be seen as describing the behavior of the modeled system in an modeling language independent manner.

As can be directly seen from the definition Kripke structures have a close relationship with automata.

The changes are the following:

- labelling is on states instead of having labels on arcs,
- there is at most one arc between two states, and
- there is no definition of final states.

Next we will continue with an example of a Kripke structure.

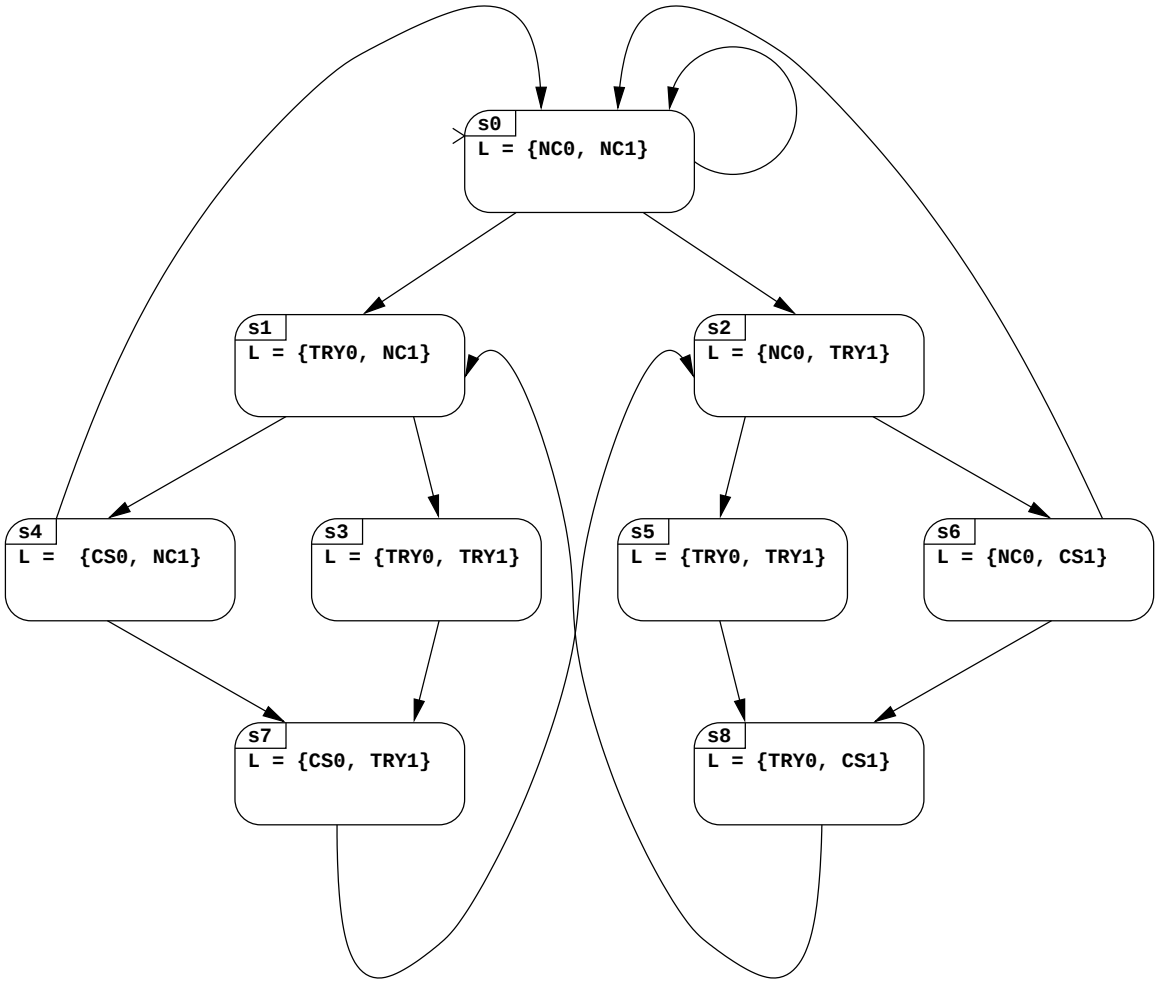

Figure 1: An Example Kripke Structure of a Mutex System

An often used trick is to actually use an automaton directly derived from the Kripke structure in model checking. We define an automaton $\mathcal{A}_M$, which accepts exactly the (finite) sequences of valuations in a **path** through the Kripke structure.

Definition 6.2 Let $M = (S, s^0, R, L)$ be a Kripke structure over a set of atomic propositions AP. Define an automaton $\mathcal{A}_M = (\Sigma, S_M, S_M^0, \rho_M, F_M)$, where

- $\Sigma = 2^{AP}$,
- $S_M = S$,
- $S_M^0 = \{s^0\}$,
- for all $s \in S, a \in \Sigma$:
  - if $L(s) \neq a$: $\rho_M(s, a) = \emptyset$,
  - if $L(s) = a$: $\rho_M(s, a) = T$, where $T \subseteq S$ is the largest set such that for all $t \in T$ it holds that $(s, t) \in R$,

  , and
- $F_M = S_M$.

An example safety property for a path in the Kripke structure is is the following:

$Spec$: The path satisfies the property if it does not contain a state having both $CR0$ and $CR1$ holding at the same time.

It easy to give a specification automaton $\mathcal{S}$ which accepts all strings of $\mathcal{A}_M$ corresponding to paths which satisfy this property.

In this case the complement of the specification $\neg Spec$ is the following:

The path satisfies the property if it contains a state in which both $CR0$ and $CR1$ hold at the same time.

It can be checked by an (even simpler) automaton $\overline{\mathcal{S}}$.

Now all paths through the Kripke structure satisfy $Spec$ iff the is no path which satisfies $\neg Spec$.

To implement this, we can thus easily obtain the product automaton $\mathcal{P} = \mathcal{A}_M \times \overline{\mathcal{S}}$, and check that indeed $L(\mathcal{P}) = \emptyset$ and thus the (safety property) expressed by $\mathcal{S}$ holds for all paths through the Kripke structure M.

For even slightly more complicated specifications expressing the specification directly as an automaton can be too complicated. This is one of the reasons temporal logics are more widely used as a specification formalisms as are automata.

Automata are, however, one of the main implementation techniques in implementing model checking algorithms for more expressive temporal logics, a subject which we will discuss next.