15-414: Bug Catching: Automated Program Verification

Lecture Notes on LTL Model Checking & Büchi Automata

Matt Fredrikson

Carnegie Mellon University
Lecture 19

1 Introduction

We've seen how to check Computation Tree Logic (CTL) formulas against computation structures. The algorithm for doing so directly computes the semantics of formulas, and makes use of the fixpoint properties of monotone functions to derive the set of states in a transition structure that satisfy the formula. We saw in a previous lecture that LTL formulas are defined over traces, of where there are infinitely many in a computation structure, so a similar approach will not work for LTL.

In this lecture, we will see how to check LTL formulas against computation structures by reducing the problem to checking whether the language defined by a finite automaton is empty. However, because the traces of a computation structure are infinite, we cannot use the familiar tools available for nondeterministic finite automata (NFAs), and instead need to define a new type of automata that can recognize infinite words. These are called Büchi automata, and we will see that they have useful properties that can be used to construct effective model checking algorithms for LTL [**?**].

2 Kripke structures & LTL

In previous lectures we introduced Kripke and computation structures, which capture the transition behavior of a computation.

Definition 1 (Kripke structure)**.** A *Kripke frame* $(W, \curvearrowright)$ consists of a set W with a transition relation $\curvearrowright \subseteq W \times W$ where $s \curvearrowright t$ indicates that there is a direct transition from s to t in the Kripke frame $(W, \curvearrowright)$. The elements $s \in W$ are also called states. A *Kripke structure* $K = (W, \curvearrowright, v, I)$ is a Kripke frame $(W, \curvearrowright)$ with a mapping $v : W \to 2^V$, where 2^V is the powerset of V assigning truth-values to all the propositional atoms in all states. Moreover, a Kripke structure has a set of initial states $I \subseteq W$.

A (computation) *path* is an infinite sequence $s_0, s_1, s_2, s_3, \ldots$ of states $s_i \in W$ such that $s_i \curvearrowright s_{i+1}$ for all i. We will always assume that the structures used in model checking are computation structures, unless otherwise noted. We can also refer to the *traces* of a Kripke structure, by evaluating each of its paths under the labeling function.

Like CTL, the temporal modalities of LTL allow us to formalize properties that involve time and sequencing. While the semantics of CTL formulas are defined over the states of a transition structure, the truth value of LTL formulas is defined over traces. Definition 2 gives the meaning of an LTL formula over a trace. Definition 3 extends the semantics to transition systems, where we require that for all traces σ obtained by running a computation structure K, $\sigma \models P$.

Definition 2 (LTL semantics (traces))**.** The truth of LTL formulas in a trace σ is defined inductively as follows:

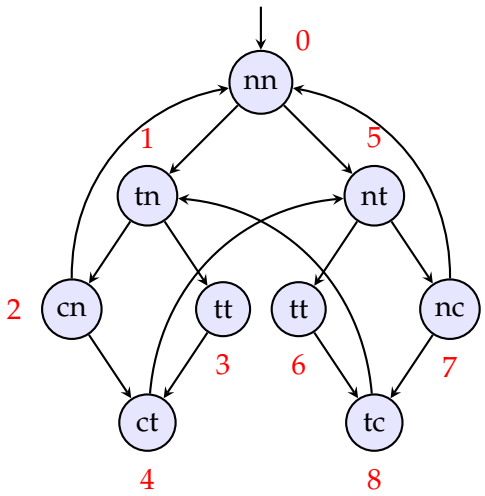
1. $\sigma \models p$ iff $\sigma_0 \models p$ for atomic propositions p provided that $\sigma_0 \neq \Lambda$
2. $\sigma \models \neg P$ iff $\sigma \not\models P$, i.e. it is not the case that $\sigma \models P$
3. $\sigma \models P \wedge Q$ iff $\sigma \models P$ and $\sigma \models Q$
4. $\sigma \models \circ P$ iff $\sigma^1 \models P$
5. $\sigma \models \Box P$ iff $\sigma^i \models P$ for all $i \geq 0$
6. $\sigma \models \Diamond P$ iff $\sigma^i \models P$ for some $i \geq 0$
7. $\sigma \models \mathbf{U} PQ$ iff there is an $i \geq 0$ such that $\sigma^i \models Q$ and $\sigma^j \models P$ for all $0 \leq j < i$

In all cases, the truth-value of a formula is, of course, only defined if the respective suffixes of the traces are defined.

Definition 3 (LTL semantics (computation structure))**.** Given an LTL formula P and computation structure $K = (W, \curvearrowright, v)$, $K \models P$ if and only if $\sigma \models P$ for all σ where $\sigma_i = v(s_i)$ for some path $s_0, s_1, s_2, \ldots$ in K.

3 LTL model checking

Let's think about how we might go about checking an LTL formula against a transition structure, using the mutual exclusion example from the last lecture. Recall that this models two abstract processes, and whether they are currently in the **n**oncritical section, **t**rying to enter the critical setion, or are in the **c**ritical section. Those atomic propositional letters are used with suffix 1 to indicate that they apply to process 1 and with suffix 2 to indicate process 2. For example the notation nt indicates a state in which $n_1 \wedge t_2$ is true (and no other propositional letters).



We can express some useful properties about the potential behavior of this computation using LTL formulas.

- The mutual exclusion safety property $\square(\neg c_1 \vee \neg c_2)$ characterizes traces where it is never the case that both processes are in the critical section at the same time. Equivalently, traces where at all times it is true that either $\neg c_1$ or $\neg c_2$.
- The liveness property $\square(t_1 \rightarrow \Diamond c_1) \wedge \square(t_2 \rightarrow \Diamond c_2)$ characterizes traces that satisfy the requirement that whenever a process tries to enter its critical section (t_i is true), it eventually succeeds (c_i becomes true).

Let's first consider the mutual exclusion safety property. In order to check that the transition structure satisfies it, we need to verify that all traces in the structure satisfy $\neg c_1 \vee \neg c_2$. As the set of traces in this structure is infinite, approaching this directly by exhaustive enumeration will not be productive. Indeed, we could proceed inductively as we have for other unbounded computations in this course.

But our experience with induction has always relied heavily on providing an invariant from which we can build a sufficiently strong inductive hypothesis. We want to develop a completely automatic technique for verifying LTL formulas, so thinking about this problem differently might yield something more suitable.

A formal language perspective Recalling that the semantics of LTL formulas are defined over traces, we can define the language $\mathcal{L}(P)$ of an LTL formula P as the set of traces that satisfy P.

Definition 4 (LTL Semantics (language over traces))**.** Let P be an LTL formula and Σ a set of atomic propositions. Then the language of P is defined as:

$$\mathcal{L}(P) = \{\sigma \in \Sigma^\omega \ : \ \sigma \models P\}$$

where Σ^ω is the set of infinite strings over Σ, and the truth relation $\models$ is defined inductively in Definition 2.

Definition 4 equates the meaning of an LTL formula with a language that describes every behavior that is allowed by the property. Viewing this set as a language, each word in the language is an infinite-length string with characters that correspond to atomic propositions. For example, the mutual exclusion property from earlier has the following word in its language:

$$\sigma = (\{\}, \{c_2\}, \{c_1\}, \{\}, \dots \text{ (repeated infinitely)})$$

In the above, we use the convention that any atomic proposition not appearing in a state is assumed to be false; so the appearance of $\{\}$ means that no atomic proposition is true, whereas $\{c_1\}$ means that c_1 is true but c_2 is false.

The following word is not in its language, because c_1 and c_2 are simultaneously true in the fourth state:

$$\sigma = (\{\}, \{c_2\}, \{c_1\}, \{c_1, c_2\}, \dots \text{ (repeated infinitely)})$$

We can also define the set of traces $\mathcal{L}(K)$ of a computation structure K, as the set of all infinite-length words over atomic propositions obtained by following transitions in K from an initial state. $\mathcal{L}(K)$ corresponds to all of the possible behaviors that K might exhibit in its execution.

Definition 5 (Language of a computation structure)**.** Let $K = (W, \curvearrowright, v)$ be a computation structure defined over a set of atomic propositions Σ. Then the language of K, denoted $\mathcal{L}(K)$, is: $\mathcal{L}(K) = \{\sigma \in \Sigma^\omega \ : \ s_0, s_1, \dots \text{ a path in } K \text{ and } \sigma_i = v(s_i)\}$.

In the computation structure given above, one such behavior (i.e. word in the language) would be:

$$\sigma = (\{n_1, n_2\}, \{n_1, t_2\}, \{n_1, c_2\}, \dots \text{ (repeated infinitely)})$$

Interpreting the LTL formula and computation structure as languages gives us a new way to think about the model checking problem. Namely, we can reason that in order for a transition structure K to satisfy formula P, it must be that every trace of K satisfies P. The languages $\mathcal{L}(P)$ gives us exactly the set of traces that satisfy P, so we have only to check that the language $\mathcal{L}(K)$ is contained in $\mathcal{L}(P)$:

$$\mathcal{L}(K) \subseteq \mathcal{L}(P) \tag{1}$$

Equation 1 equivalent to saying that all of the behaviors of K are among the set of behaviors that are allowed by P.

Checking by complement How can we check whether Equation 1 holds for a given K and P? Suppose for the moment that $\mathcal{L}(K)$ and $\mathcal{L}(P)$ were regular languages containing only finite words. Then we could exploit the fact that regular languages are closed under intersection and complementation, in addition to the following fact (see [**?**] or for a proof):

$$\mathcal{L}(K) \subseteq \mathcal{L}(P) \text{ if and only if } \mathcal{L}(K) \cap \overline{\mathcal{L}(P)} = \emptyset \tag{2}$$

$\overline{\mathcal{L}(P)}$ is the complement of $\mathcal{L}(P)$, i.e., the set of all behaviors that are not allowed by P. We can check that Equation 2 matches the intuition developed so far: if $\mathcal{L}(K) \cap \overline{\mathcal{L}(P)}$ is empty, then there are no behaviors of K that are *not* allowed by P. Removing the double negative, *all* behaviors of K are allowed by P.

Assuming we have the finite-state machine corresponding to a regular language, checking whether that language is empty is a reachability problem **[?, ?]**: we simply look for a path through the automaton from an initial state to an accepting state. This suggests the following algorithm for checking property P against transition structure K (assuming both are equivalent to regular languages):

1. Construct finite-state machines A_K and $A_{\overline{P}}$ corresponding to $\mathcal{L}(A)$ and $\overline{\mathcal{L}(P)}$, respectively. We know that $A_{\overline{P}}$ exists because regular languages are closed under complementation.

2. Use the fact that regular languages are closed under intersection to compute $A_{K \cap \overline{P}}$ from A_K and $A_{\overline{P}}$.

3. Check whether $\mathcal{L}(K) \cap \overline{\mathcal{L}(P)}$ is empty by looking for a path in $A_{K \cap \overline{P}}$ from an initial state to an accepting state.
   a) If $\mathcal{L}(K) \cap \overline{\mathcal{L}(P)} = \emptyset$, then conclude that $\mathcal{L}(K) \subseteq \mathcal{L}(P)$ so K satisfies P ($K \models P$).
   b) If $\mathcal{L}(K) \cap \overline{\mathcal{L}(P)} \neq \emptyset$, then conclude that $K \not\models P$. Any word in $\mathcal{L}(K) \cap \overline{\mathcal{L}(P)}$ corresponds to a counterexample of P, i.e., a trace exhibiting a behavior in K that is not allowed by P.

This procedure is appealing for several reasons. It is completely automatic, and reduces model checking to a reachability problem over the graph of an automata. In cases where the transition structure does not satisfy the property in question, there is a simple procedure for extracting counterexamples that witness this fact; such counterexamples can be useful in practice for diagnostic reasons by highlighting behaviors that violate the property.

Of course, we can't actually use this procedure to check LTL formulas against computation structures because we know that $\mathcal{L}(P)$ and $\mathcal{L}(K)$ are not regular languages—their words are infinite, and can't be recognized by finite state machines.

4 Automata on Infinite Words

In order to recover a model checking procedure like the one described in the one described in the previous section, we look to automata that accept languages of infinite words. Nondeterministic Büchi automata (NBAs) are a variant of nondeterministic finite automata (NFAs) that do exactly this.

Definition 6 (Nondeterministic Büchi Automaton (NBA))**.** A nondeterministic Büchi automaton A is a tuple $A = (Q, \Sigma, \delta, Q_0, F)$ where:

1. Q is a **finite** set of states.
2. Σ is an alphabet.
3. $\delta : Q \times \Sigma \to \wp(Q)$ is a transition function.
4. $Q_0 \subseteq Q$ is a set of initial states
5. $F \subseteq Q$ is a set of accepting states, which we sometimes call the *acceptance set*.

A run for (infinite) trace $\sigma = \sigma_0, \sigma_1, \sigma_2, \ldots$ is an infinite sequence of states $q_0, q_1, q_2, \ldots$ in Q such that $q_0 \in Q_0$ and $q_{i+1} \in \delta(q_i, \sigma_i)$ for all $i \geq 0$. A run $q_0, q_1, q_2, \ldots$ is accepting if $q_i \in F$ for **infinitely many indices** $i \geq 0$. The language of A is:

$$\mathcal{L}(A) = \{\sigma \in \Sigma^\omega \ : \ \text{there exists an accepting run for } \sigma \text{ in } A\}$$

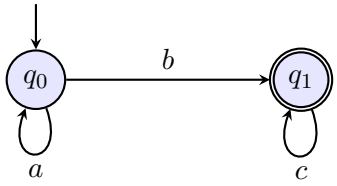
In the above, Σ^ω is the set of all infinite words over alphabet symbols in Σ.

Notice that in terms of syntax, there is no distinction between NBAs and NFAs: both have a finite number of states, an alphabet, a transition function, and a subset of initial and accepting states. The transition relation in a NBA works in exactly the same way as in a NFA, i.e., by consulting the "row" for the current state and alphabet symbol to determine which state (of potentially many) to visit next.

The difference is in the semantics. NBAs accept infinite words, so it is meaningless to consider whether a run ends in an accepting state (as in the case of NFAs) because there is no end to an infinite run. Rather, the semantics of NBAs require than an accepting run visit the acceptance set F **infinitely often**. This might seem quite demanding at first, but because the set of states Q is finite, any infinite run must visit *some* non-empty set of states $Q' \subseteq Q$ infinitely often. The acceptance criterion simply asks whether Q' has a non-empty intersection with F.

As a convenient shorthand, we will use Boolean combinations of atomic propositions to label transitions. So if $\Sigma = \wp(\{a, b\})$ then a transition labeled $a \vee b$ stands for three separate transitions: one labeled by $\{a\}$, another labeled by $\{b\}$, and the third by $\{a, b\}$.

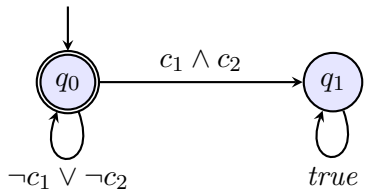
Notice that Definition 6 does not require that δ give each state a direct successor, or impose any form of totality on it. This might seem strange in light of the corresponding requirement for computation structures, as NBAs intend to capture infinite behaviors just like the former. However, there is no contradiction here. Consider the following example, which accepts all infinite strings of $\{a, b, c\}$ that begin with a finite number of a's, followed by a single b, following by an infinite number of c's.



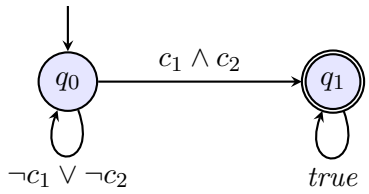
From state q_0, there do not exist any transitions on symbol c. So is the word $acbcccc\ldots$ in the language of this NBA? Looking at the semantics given in Definition 6, we see that it is not. In order to be in the language, there must exist an accepting run, and there is no way to run this NBA on the word $acbcccc\ldots$ because it "falls off" of the transition relation.

Examples Going back to our original goal of checking the safety and liveness properties of the mutual exclusion example, recall the formula $\Box(\neg c_1 \vee \neg c_2)$. We can represent this property using a NBA, by setting the alphabet Σ to be $\wp(\text{atomic propositions}) = \wp(\{c_1, c_2, n_1, n_2, t_1, t_2\})$.

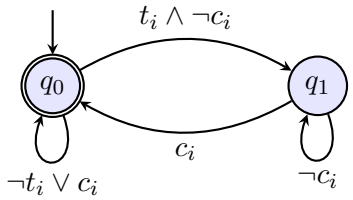
Returning to the automaton for $\Box(\neg c_1 \vee \neg c_2)$, the single initial state q_0 of the automaton is also the acceptance set, and there is a self-transition on this initial state labeled $\neg c_1 \vee \neg c_2$. The second (and only other) state q_1 is not in the acceptance set, and is reachable from q_0 on $c_1 \wedge c_2$. Finally, there must be a self-loop on q_1 for any alphabet symbol (i.e., $true$), because once the mutual exclusion **invariant** is violated by $c_1 \wedge c_2$, there is no way to "repair" the trace so that it satisfies the property. The transition diagram is shown below.



We can also build an automaton for the complement of this property, which corresponds to the set of all "bad" behaviors that violate the mutual exclusion property. In this case, the complement is easily obtained by swapping the states in the acceptance set $\{q_0\}$ with their complement $\{q_1\}$. This is due to the fact that the automaton is actually deterministic. For general NBA, complementation is not so straightforward [**?**], but we will return to this inconvenience later on.



Looking at another example, let's build an NBA for $\Box(t_1 \rightarrow \Diamond c_1) \wedge \Box(t_2 \rightarrow \Diamond c_2)$. Because either side of the conjunction is symmetrical with the other, we will show one automaton for $\Box(t_i \rightarrow \Diamond c_i)$ that can be instantiated twice to arrive at the full NBA.



This NBA begins in its accepting state, and stays there as long as process i does not try to enter its critical section (or it tries to enter, and succeeds immediately in the same state). If the process tries to enter its critical section and does not immediately succeed ($t_i \wedge \neg c_i$), then the NBA transitions to a non-accepting state and stays there as long as the process doesn't enter the critical section ($\neg c_i$). Finally, if the process enters its critical section (c_i), the automaton transitions back to its initial accepting state.

Computation structures and Büchi automata We are moving towards a language-theoretic solution to the LTL model checking problem. Recall that the first steps in the case of regular languages was to obtain automata that represent the languages of the computation structure and LTL property. We've seen an example of how to convert an LTL property into a NBA, and we'll return to a more general solution for converting any LTL formula to NBA later. For now, let's convince ourselves that a given computation structure $K = (W, \curvearrowright, v)$ with initial states W_0 can be represented with NBA.

Theorem 7. *Let $K = (W, \curvearrowright, v)$ be a computation structure with initial states W_0 over atomic predicates AP. Then the the nondeterministic Büchi automaton A_K given by the following criterion satisfies $\mathcal{L}(A_K) = \mathcal{L}(K)$,*

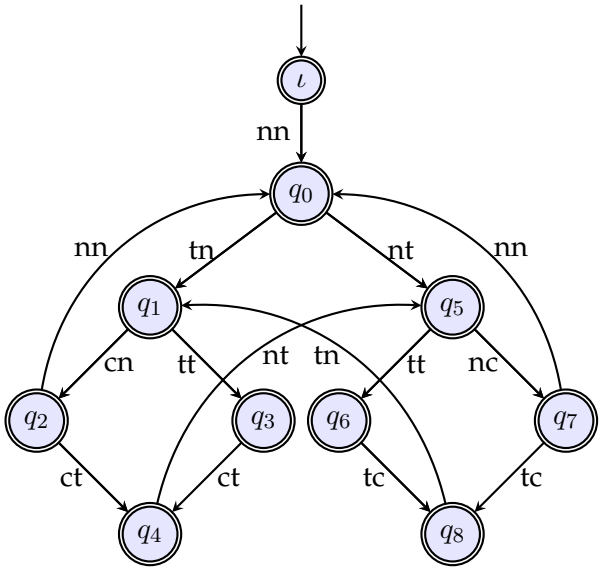
$$A_K = (Q = W \cup \{\iota\}, \Sigma = \wp(AP), \delta, Q_0 = \{\iota\}, F = W \cup \{\iota\})$$

where $q' \in \delta(q, \sigma)$ iff $q \curvearrowright q'$ and $v(q', \sigma)$, and $q \in \delta(\iota, \sigma)$ whenever $q \in Q_0$ and $v(q, \sigma)$.

Theorem 7 says that a computation structure K is converted to a NBA A_K with the following steps:

1. The states of A_K are identical to those of K, except a new initial state ι not appearing in K is added. ι is the only initial state of A_K.
2. The alphabet of A_K is the powerset of the atomic propositions AP used to define K.
3. The transition function δ of A_K includes all of the state transitions appearing in K. The transition symbols for δ correspond to the atomic propositions assigned by v to the post state of each element of $\curvearrowright$. Moreover, δ gives transitions from ι to every initial state $q \in Q_0$, again using the transition symbols from $\wp(AP)$ that v assigns to the corresponding $q \in Q_0$.
4. The acceptance set of A_K corresponds to all of the states $W \cup \{\iota\}$. This is due to the fact that *all* runs of K that obey the transition relation are in $\mathcal{L}(K)$, so any trace that doesn't "fall off" of A_K is in $\mathcal{L}(A_K)$.

As an example, below we show the NBA corresponding to our running mutual exclusion computation structure. Notice that even though there is only one initial state in the original computation structure, it has still been replaced in the NBA with the distinguished state ι. While it may not seem as though we have gained anything by doing this, because we label transitions on the NBA with the atomic propositions of the post state from the computation structure, there must be an incoming transition to this state in the NBA so that the first symbol from words appearing in $\mathcal{L}(K)$ is processed consistently with the rest.



Closure under intersection As it turns out, NBAs are closed under intersection just as are their NFA counterparts over finite words. The proof of this fact is given directly by construction of a product automaton that accepts exactly the language of the intersection of its components **[?, ?]**.

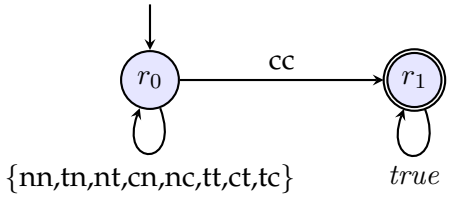
While this construction is straightforward, one does need to be careful about the acceptance set of the product NBA. In particular, when taking the product of $A_1 = (Q_1, \Sigma_1, \delta_1, Q_1^0, F_1)$ and $A_2 = (Q_2, \Sigma_2, \delta_2, Q_2^0, F_2)$, we need to ensure that words accepted by $A_1 \cap A_2$ go through states corresponding to F_1 and F_2 an infinite number of times. To accomplish this, the product construction splits states into three distinct parts $0, 1, 2$ function intuitively as follows:

1. The product construction has all its initial states in part 0.
2. When entering a state corresponding to F_1, the product moves to a state in part 1.
3. When entering a state corresponding to F_2, the product moves to a state in part 2.
4. When the product is in a state from part 2, and enters a state not in F_2, transition back to a state in part 0.

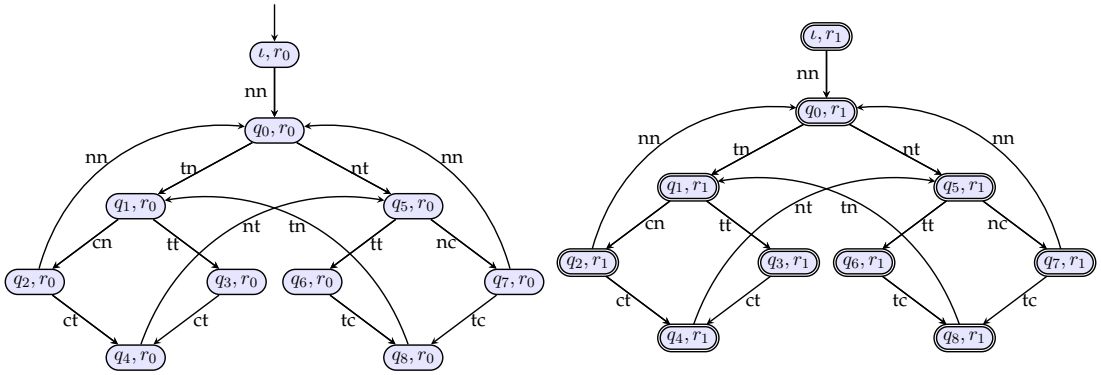
Further details of this construction are given in [**?**]. For the purposes of our goals, we can use a simplified product construction that relies on the fact that the NBA obtained from a computation structure has an acceptance set corresponding to its entire state space.

Theorem 8. *Given two nondeterministic Büchi automata* $A_1 = (Q_1, \Sigma, \delta_1, Q_1^0, Q_1)$ *and* $A_2 = (Q_2, \Sigma, \delta_2, Q_2^0, F)$*, the product* $A_{1\cap 2} = (Q_1 \times Q_2, \Sigma, \delta', Q_1^0 \times Q_2^0, Q_1 \times F)$*, where* $(q_1', q_2') \in \delta'((q_1, q_2), \sigma)$ *iff* $(q_i') \in \delta(q_i, \sigma)$ *for* $i = 1, 2$*, satisfies* $\mathcal{L}(A_{1\cap 2}) = \mathcal{L}(A_1) \cap \mathcal{L}(A_2)$.

To see Theorem 8 in action, let's return to the task of checking the mutual exclusion safety property on the NBA corresponding to the mutual exclusion computation structure. We'll start be renaming the states in the NBA for the safety property, and updating the transition labels to make them consistent with those used in the computation structure's NBA.



We can now proceed with the intersection. The resulting automaton shown below consists of two disconnected components, the first corresponding to states containing r_0 and the second to states containing r_1. They are disconnected because in the property NBA, the only transition between r_0 and r_1 is labeled **cc**. However, the computation NBA has no transitions labeled **cc**, and the δ' from Theorem 8 requires corresponding transitions in *both* constituent NBA.



Importantly, the initial state in the product is one containing r_0, and the acceptance set consists entirely of those containing r_1. It is evident that the language of this NBA is the empty set, which confirms our expectation that the original computation structure satisfies the mutual exclusion safety property.

Checking emptiness The previous example was easy to check "visually" by inspection, because none of the accepting states were reachable from the single initial state. In general of course this heuristic will not apply, so we need a more general algorithm for determining whether the product NBA corresponds to the empty language.

Consider an NBA A and accepting run $\rho = q_0, q_1, \ldots$. Because ρ is accepting, it contains infinitely many accepting states from F, and moreover, because $F \subseteq Q$ is finite, there is some suffix ρ' of ρ such that every state on it appears infinitely many times. In order for this to happen each state in ρ' must be reachable from every other state in ρ', which means that these states comprise a strongly-connected component in A. From this we can conclude that any strongly connected component in A that *(1)* is reachable from the initial state, and *(2)* contains at least one accepting state, will generate an accepting run of the automaton.

Whenever such a strongly-connected component exists in the NBA, there will necessarily be a cycle from some accepting state back to itself; given a strongly-connected component with an accepting state, it is always possible to find such a cycle, and the converse clearly holds. So given a product automaton as described in the previous sections, we can perform model checking using any cycle detection algorithm such as Tarjan's depth-first search **[?]**. This runs in time $O(|Q| + |\delta|)$, but is sometimes not as efficient in practice as other alternatives that we will cover in the next lecture.

References