**15-414: Bug Catching: Automated Program Verification**

# Lecture Notes on LTL Model Checking & Büchi Automata

Matt Fredrikson

Carnegie Mellon University

Lecture 17

## 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 [Var86].

## 2 Review: LTL

In the previous lecture, we introduced Linear Temporal Logic (LTL). The temporal modalities of LTL allow us to formalize properties that involve time and sequencing, where the truth value of an LTL formula is defined over traces, or potentially infinite sequences of symbols from an alphabet of states. Definition 1 gives the meaning of an LTL formula over a trace. Definition 4 extends the semantics to transition systems, where we require that for all traces $\sigma$ obtained by running a computation structure $K$, $\sigma \models P$.

**Definition 1** (LTL semantics (traces))**.** The truth of LTL formulas in a trace $\sigma$ is defined inductively as follows:

1. $\sigma \models F$ iff $\sigma_0 \models F$ for state formula $F$ 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. $\sigma \models P \land Q$ iff $\sigma \models P$ and $\sigma \models Q$
4. $\sigma \models \mathbf{X}P$ 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. $\sigma \models P\mathbf{U}Q$ 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.

## 3 Transition structures of computations

In the previous lecture we defined a semantics for the familiar simple imperative language that equates programs with sets of traces over states. This generalized the previous relational semantics that we used when reasoning about contracts, and allowed us to evaluate LTL formulas over runs of programs. Another way of formalizing the semantics of a program, or for that matter any arbitrary computation, is to define a structure that describes the way in which the computation can transition between states. We can even recover the trace semantics from the previous lecture from this, by assigning some set of initial states and collecting the set of traces that one gets by following the transition structure repeatedly.

**Definition 2** (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* $K = (W, \curvearrowright, v)$ is a Kripke frame $(W, \curvearrowright)$ with a mapping $v : W \to \Sigma \to \{\mathit{true}, \mathit{false}\}$ assigning truth-values to all the propositional atoms in all states.

Note that this definition does not explicitly account for initial states. We will generally assume that all states are possible initial states, and if we need to restrict this in some specific cases, we will be sure to make a note of which states are initial. Given a program $\alpha$, we can intuitively see how it is possible to define a Kripke structure whose traces correspond to $\tau(\alpha)$. But note that the relational program semantics $[\![\alpha]\!]$ between initial and final states in Lecture 3 is also an example of a Kripke structure; checking that this is the case is a good exercise to help familiarize yourself with the above definition.

Kripke structures impose no requirements on the totality of transitions or the finiteness of the state space, but it is sometimes helpful to assume such restrictions. *Computation structures* (Definition 3 below), refine Kripke structures by requiring the state

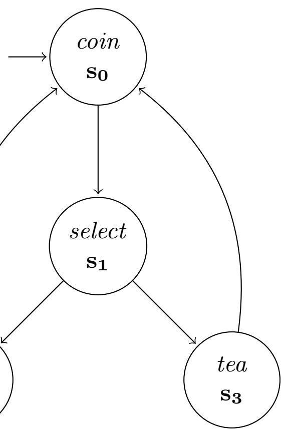


Figure 1: Computation structure describing the operation of a vending machine.

space to be finite and each state to have at least one successor. These conditions make it possible to define model checking algorithms for unbounded (potentially infinite) computations.

**Definition 3** (Computation structure)**.** A Kripke structure $K = (W, \curvearrowright, v)$ is called a *computation structure* if $W$ is a finite set of states and every element $s \in W$ has at least one direct successor $t \in W$ with $s \curvearrowright t$. A (computation) *path* in a computation structure 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 $i$. The mapping $v$ is the same as in Definition 2.

Finally, we can now define LTL semantics over computations structures rather than individual traces. Intuitively, a formula $P$ is true for a computation structure $K$ iff $\sigma \models P$ for all paths $\sigma$ in $K$.

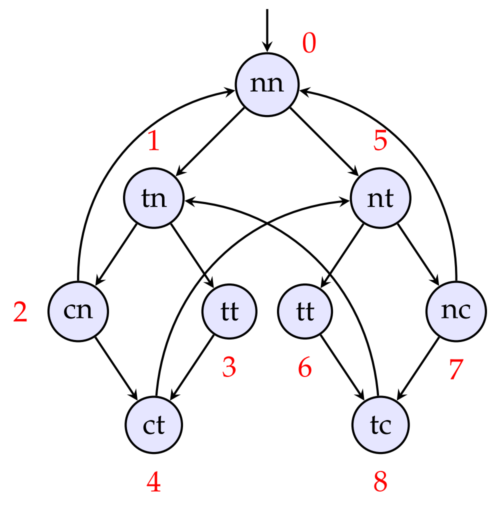
**Definition 4** (LTL semantics (computation structure))**.** Given an LTL formula $P$ and computation structure $K = (W, \curvearrowright, v)$, $K \models P$ if and only if $\sigma \models P$ for all $\sigma$ where $\sigma_i = v(s_i)$ for some path $s_0, s_1, s_2, \ldots$ in $K$.

Some examples of these structures are useful in developing intuition. The set of states $W$ represented in Figure 1 are $W = \{s_0, s_1, s_2, s_3\}$. The propositional atoms $\Sigma$ that appear in those states are $\Sigma = \{\text{coin,select,coffee,tea}\}$. Here we do assume an initial state $I = \{s_0\}$. The mapping $v$ and transition relation are given as follows:

$$\begin{aligned} s_0 &\to \{\text{coin} \to \mathit{true}\} \\ s_1 &\to \{\text{select} \to \mathit{true}\} \\ s_2 &\to \{\text{coffee} \to \mathit{true}\} \\ s_3 &\to \{\text{tea} \to \mathit{true}\} \end{aligned} \qquad \begin{aligned} s_0 &\curvearrowright s_1 \\ s_1 &\curvearrowright s_2 \\ s_1 &\curvearrowright s_3 \\ s_2 &\curvearrowright s_0 \\ s_3 &\curvearrowright s_0 \end{aligned}$$

Note that we only shown the propositional atoms that are assigned the truth value $\mathit{true}$ but the remaining atoms would be assigned truth value $\mathit{false}$.

Temporal logic is particularly helpful in verifying properties of concurrent systems. The following computation structure represents a system of two concurrent processes, each of which is either executing in a **n**oncritical section, **t**rying to enter the critical setion, or are in the **c**ritical section These atomic propositional letters are used with suffix 1 to indicate that they apply to process 1 and with suffix 2 to indicate process 2. So for example, the notation $nt$ indicates a state in which $n_1 \wedge t_2$ is true (and no other propositional letters), meaning that process 1 is in its noncritical section, and process 2 is trying to enter its noncritical section.



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

- The mutual exclusion *safety* property $\Box(\neg c_1 \vee \neg c_2)$ 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 $\Box(t_1 \rightarrow \Diamond c_1) \wedge \Box(t_2 \rightarrow \Diamond c_2)$ 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).

More generally, **safety properties** impose constraints which stipulate that something "bad" never happens; in the example above, the "bad" thing is having both processes in the critical section at the same time. **Liveness properties** specify that something "good" will always happen eventually; in the above example, the "good" thing is entering the critical section eventually after having tried to do so.

## 4 LTL model checking

Continuing with the most recent example of two concurrent processes, let's take a closer look at 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 $\neg c_1 \vee \neg c_2$ at all times. 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 we will take a different approach.

**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 5** (LTL Semantics (language over traces))**.** Let $P$ be an LTL formula and $\Sigma$ a set of atomic propositions. Then the language of $P$ is defined as:

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

where $\Sigma^\omega$ is the set of infinite strings over $\Sigma$, and the truth relation $\models$ is defined inductively in Definition 1.

Definition 5 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 sets of atomic propositions. For example, the mutual exclusion property from earlier has the following word in its language:

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

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 the language of $\Box(\neg c_1 \vee \neg c_2)$, because $c_1$ and $c_2$ are simultaneously true in the fourth state:

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

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

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

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

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 [BKL08] 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 [BKL08, CGP99]: 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.

## 5 Automata on Infinite Words

In order to recover a model checking procedure like 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 7** (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. $\Sigma$ 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:

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

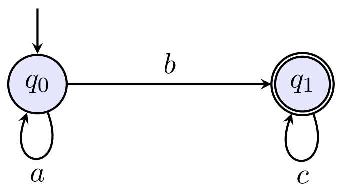
In the above, $\Sigma^\omega$ is the set of all infinite words over alphabet symbols in $\Sigma$.

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 $\{a, b\}$.

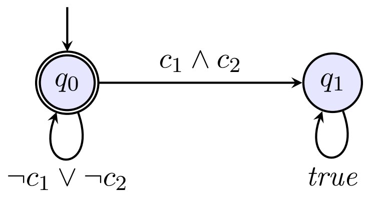
Notice that Definition 7 does not require that $\delta$ give each state a direct successor, or impose any form of totality on it. This might seem strange in light of the corresponding requirement for computation structures, as NBAs intend to capture infinite behaviors just like the former. However, there 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.



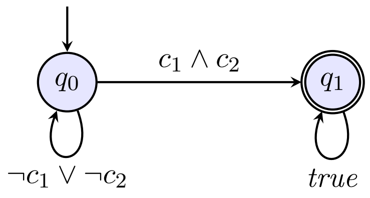
From state $q_0$, there do not exist any transitions on symbol $c$. So is the word $acbcccc\ldots$ in the language of this NBA? Looking at the semantics given in Definition 7, we see that it is not. In order to be in the language, there must exist an accepting run, and there is no way to run this NBA 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 $\Sigma$ to be $\wp(\text{atomic propositions}) = \wp(\{c_1, c_2, n_1, n_2, t_1, t_2\})$.

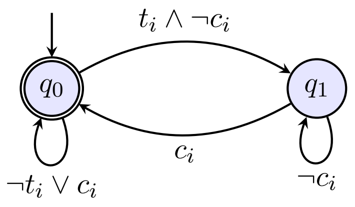
Returning to the automaton for $\Box(\neg c_1 \vee \neg c_2)$, the single initial state $q_0$ of the automaton is also the acceptance set, and there is a self-transition on this initial state labeled $\neg c_1 \vee \neg c_2$. The second (and only other) state $q_1$ is not in the acceptance set, and is 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 [B̈62], 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 $\Box(t_i \rightarrow \Diamond c_i)$.



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 8.** *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 $q \in W_0$ and $v(q, \sigma)$.*

Theorem 8 says that a computation structure $K$ is converted to a NBA $A_K$ with the following steps:

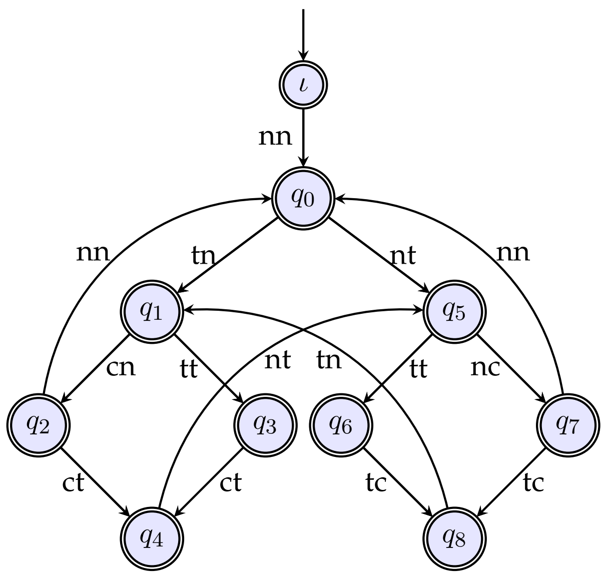
1. The states of $A_K$ are identical to those of $K$, except a new initial state $\iota$ not appearing in $K$ is added. $\iota$ is the only initial state of $A_K$.

2. The alphabet of $A_K$ is the powerset of the atomic propositions $AP$ used to define $K$.

3. The transition function $\delta$ of $A_K$ includes all of the state transitions appearing in $K$. The transition symbols for $\delta$ correspond to the atomic propositions assigned by $v$ to the post state of each element of $\curvearrowright$. Moreover, $\delta$ gives transitions from $\iota$ to every initial state in $W_0$, again using the transition symbols from $\wp(AP)$ that $v$ assigns to the corresponding $q \in W_0$.

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 $\iota$. 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.



## References

[B6̈2] J. R. Büchi. On a decision method in restricted second order arithmetic. In *Proceedings International Congress on Logic, Method, and Philosophy of Science*. Standford University Press, 1962.

[BKL08] Christel Baier, Joost-Pieter Katoen, and Kim Guldstrand Larsen. *Principles of Model Checking*. MIT Press, 2008.

[CGP99] Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model Checking*. MIT Press, Cambridge, 1999.

[Var86] Moshe Y. Vardi. On epistemic logic and logical omniscience. In Joseph Y. Halpern, editor, *Proceedings of the 1st Conference on Theoretical Aspects of Reasoning about Knowledge, Monterey, CA, USA, March 1986*, pages 293–305. Morgan Kaufmann, 1986.