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

# Lecture Notes on Emptiness Checking, LTL Büchi Automata

Matt Fredrikson André Platzer

Carnegie Mellon University

Lecture 18

## 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 [VW86].

## 2 Review: Transition structures, LTL

Several lectures ago, we introduced Linear Temporal Logic (LTL). 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 1 gives the meaning of an LTL formula over a trace. Definition 2 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. $\sigma \models p$ 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. $\sigma \models P\mathbf{U}Q$ 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.

**Definition 2** (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 $K$.

**Definition 3** (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 4** (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: $\mathcal{L}(K) = \{\sigma \in \Sigma^\omega \ : \ s_0, s_1, \ldots$ a path in $K$ and $\sigma_i = v(s_i)\}$.

By defining languages for LTL formulas and computation structures, we can case the LTL model checking problem as one of language inclusion.

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

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

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

We then defined a type of automaton that characterizes languages with infinite words.

**Definition 5** (Nondeterministic Büchi Automaton (NBA))**.** A nondeterministic Büchi automaton $A$ is a tuple $A = (Q, \Sigma, \delta, Q_0, F)$ where:

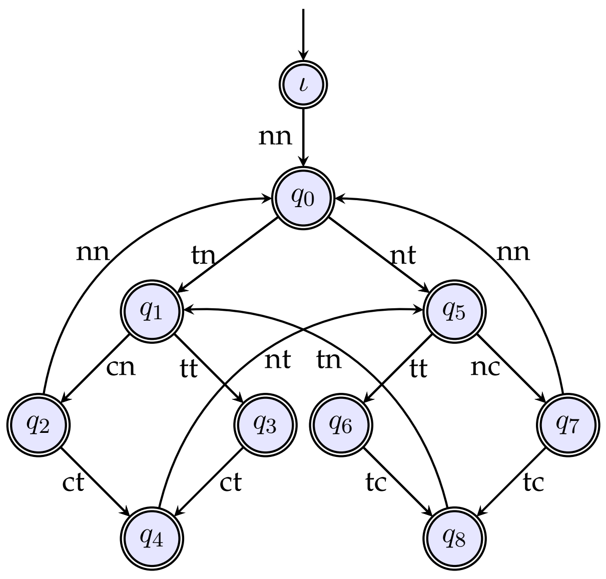
1. $Q$ is a **finite** set of states.
2. $\Sigma$ is an alphabet.
3. $\delta : Q \times \Sigma \to \wp(Q)$ is a transition function.
4. $Q_0 \subseteq Q$ is a set of initial states
5. $F \subseteq Q$ is a set of accepting states, which we sometimes call the *acceptance set*.

A run for (infinite) trace $\sigma = \sigma_0, \sigma_1, \sigma_2, \dots$ is an infinite sequence of states $q_0, q_1, q_2, \dots$ 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 $q_0, q_1, q_2, \dots$ 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 $\Sigma$.

**Running example** We demonstrated the concepts and ideas from the previous lecture on a running example of a mutual exclusion protocol. The computation corresponded to the following NBA.

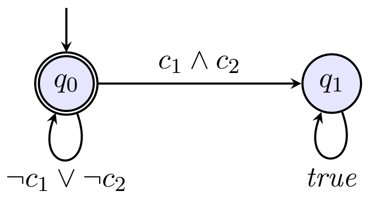


We wrote two useful properties for this computation.

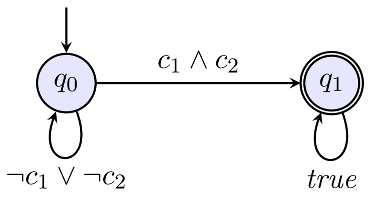
- The mutual exclusion safety property $\square(\neg c_1 \lor \neg c_2)$ characterizes traces where it is never the case that both processes are in the critical section at the same time. Equivalently, traces where at all times it is true that either $\neg c_1$ or $\neg c_2$.

- The liveness property $\square(t_1 \to \lozenge c_1) \land \square(t_2 \to \lozenge c_2)$ 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).

We found that the safety property gives the following NBA.



Its complement was easy to obtain as well.



In order to determine whether the mutual exclusion protocol models this property, we need to construct an automaton for the intersection of their languages.

## 3 Intersecting Büchi automata with Kripke structures

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 [CGP99, BKL08].

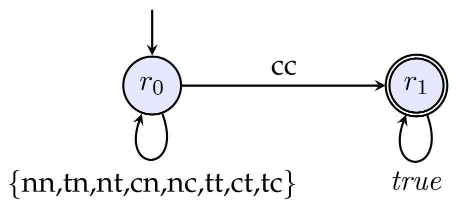
While this construction is straightforward, one does need to be careful about the acceptance set of the product NBA. In particular, when taking the product of $A_1 = (Q_1, \Sigma_1, \delta_1, Q_1^0, F_1)$ and $A_2 = (Q_2, \Sigma_2, \delta_2, Q_2^0, F_2)$, we need to ensure that words accepted by $A_1 \cap A_2$ 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.

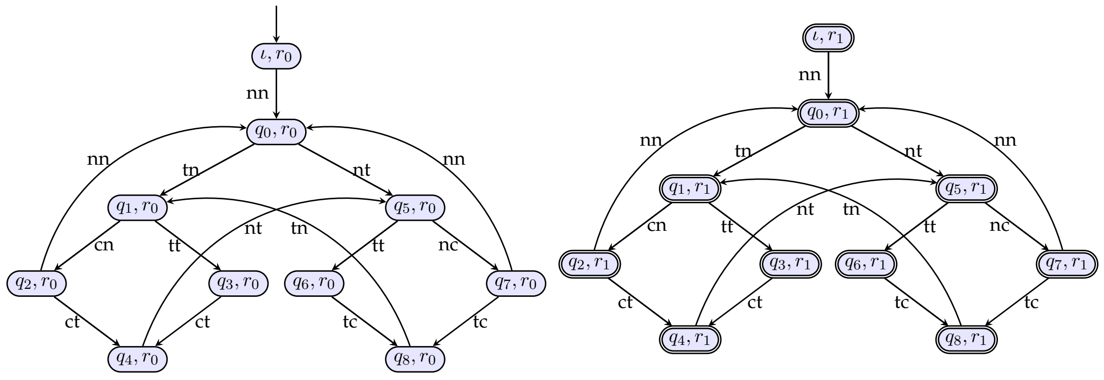
Further details of this construction are given in [CGP99]. For the purposes of our goals, we can use a simplified product construction that relies on the fact that the NBA obtained from a computation structure has an acceptance set corresponding to its entire state space.

**Theorem 6.** *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* $(q_i') \in \delta_i(q_i, \sigma)$ *for* $i = 1, 2$*, satisfies* $\mathcal{L}(A_{1\cap 2}) = \mathcal{L}(A_1) \cap \mathcal{L}(A_2)$*.*

To see Theorem 6 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 by 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 $\delta'$ from Theorem 6 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.

## 4 Emptiness checking via cycle detection

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 $\rho$ is accepting, it contains infinitely many accepting states from $F$, and moreover, because $F \subseteq Q$ is finite, there is some suffix $\rho'$ of $\rho$ such that every state on it appears infinitely many times. In order for this to happen each state in $\rho'$ must be reachable from every other state in $\rho'$, 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. Exploiting this observation, we see that it suffices to use any algorithm for detecting strongly-connected components, such as Tarjan's depth-first search algorithm [Tar72], for LTL model checking.

The asymptotic complexity of Tarjan's algorithm is $O(|Q| + |\delta|)$, i.e., linear in the number of states and transitions. We can't expect to do better than this in the worst case, but in practice it is not an ideal solution because it always constructs the entire product automaton in memory, and returns each strongly-connected component. This functionality is more than we need for LTL model checking, because it suffices to find just one counterexample to demonstrate that the computation structure fails to satisfy a property.

A notable alternative approach is based on nested depth-first cycle detection algorithm [CVWY92]. To see why detecting cycles solves LTL model checking, observe that whenever a reachable strongly-connected component with an accepting state exists in the product 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.

The nested depth-first search routine for emptiness checking, `isempty` proceeds by enumerating over all of the initial states, calling the `outerdfs` algorithm on each one. If a cycle is reachable from one of these states, then `outerdfs` will raise a `Found` exception terminating execution early. If no exception is raised throughout this enumeration, then `isempty` returns `false` to signify that no such cycle exists.

```
let isempty A =
  let (Q, Σ, δ, Q₀, F) = A in
  foreach q₀ in Q₀ do
    try
      outerdfs q₀ Nil A
    with Found -> true
  done;
  false
```

The first depth-first search implemented in `outerdfs`, which takes a state from which to begin the search, a list of states that have already been visited in this phase of the search, and the original automaton.

```
let rec outerdfs q visited A =
  let (Q, Σ, δ, Q₀, F) = A in
  let visited' = Cons q visited in
  foreach q' in δ(q) do
    if not (mem q visited') then (outerdfs q' visited' A);
  done;
  if (mem q F) then (innerdfs q visited' Nil A)
```

`outerdfs` proceeds recursively in typical depth-first fashion, exploring all immediate successors that have not already been visited by the outer search. When it is ready to backtrack, it calls the nested `innerdfs` if the state currently being considered is accepting.

Finally, the second DFS search `innerdfs` takes a state from which to begin searching, the list of states visited by the outer DFS that invoked it, the list of states visited by the current inner DFS, and the automaton.

```
let rec innerdfs q outervisited innervisited A =
  let (Q, Σ, δ, Q₀, F) = A in
  let innervisited' = Cons q innervisited in
  foreach q' in δ(q) do
    if (mem q' outervisited) then
      raise Found
    else
      if not (mem q' innervisited') then
        (innerdfs q' outervisited innervisited' A);
  done
```

If `innerdfs` ever encounters a state visited by the invoking `outerdfs`, then it terminates the search with the result `Found`. This is correct, because `innerdfs` is only called from accepting states; because it reached a previously-visited state following transitions, there must be a reachable cycle back to the accepting state from which `innerdfs` was called.

We can construct a counterexample to the emptiness claim by extracting a finite prefix from an initial state to a cycle by traversing the visited list of `outerdfs`. Let $q_1$ be the state from which the call to `innerdfs` was started, and $q_2$ the state that terminates it. Then the visited list of `innerdfs` contains cyclical path from $q_1$ to $q_2$. Concatenating this cycle to the finite prefix yields a cycle through an accepting state that is rechable from an initial state, and serves as the counterexample.

Otherwise, `innerdfs` continues again in recursive depth-first fashion avoiding work by not descending on states which have already been visited by this level of the DFS. If it never encounters a state visited by `outerdfs`, then it returns without raising an exception.

## 5 Translating LTL into Büchi automata

Let's look at how we can convert arbitrary LTL formulas into Büchi automata that accept the same language. The approach that we take, which is due to [GPVW96], uses some ideas that are related to those we observed when checking CTL formulas. In particular, just as we could write expansion axioms for CTL formulas that reduced reasoning about their satisfaction to local reasoning about the current state, along with eventual reasoning about future states, we can do so for LTL formulas as well. We have the following axioms.

$$P_1 \mathbf{U} P_2 \leftrightarrow P_2 \lor (P_1 \land \circ(P_1 \mathbf{U} P_2)) \tag{3}$$

$$\Diamond P \leftrightarrow P \lor \circ\Diamond P \tag{4}$$

$$\Box P \leftrightarrow P \land \circ\Box P \tag{5}$$

To perform the LTL-Büchi conversion, we will first construct a directed graph whose nodes contain information about the truth value of formulas. To keep track of this information, each node is labeled with three sets of formulas that we will call **old**, **now**, and **next**. The nodes of this graph will become the states in the final NBA, and the formulas in these sets correspond to properties that must be satisfied by traces that enter the corresponding nodes. The distinction between the sets is described as follows.

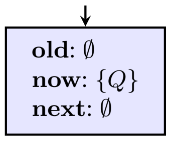
**old:** Formulas that have already been processed. As the conversion proceeds, **old** will grow to contain all of the formulas relevant to the state corresponding to this node.

**now:** Formulas that have not yet been processed. Eventually, all of the formulas in **now** will be processed, and subsequently moved to **old**.

**next:** Formulas that must be satisfied by direct successors of states satisfying the properties in **old**.

We also keep track of the set of incoming edges at each node.

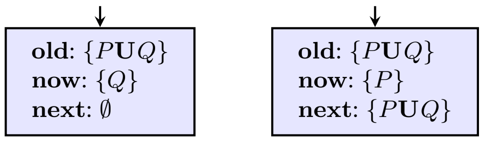
We'll first see how the conversion works by considering the example formula $R \equiv P\mathbf{U}Q$. To begin, we create a single node, setting its incoming edges to the special symbol `init`, as well as **now** $= \{P\mathbf{U}Q\}$, and **old** $=$ **next** $= \emptyset$.



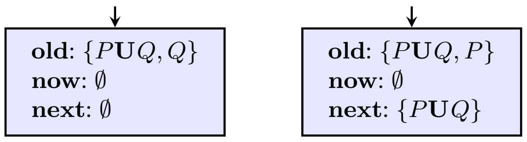
Then at each step, we check the current set of nodes to see if there are remaining formulas to process in **now**. If there are, then we select a formula from **now** to remove from the set and proceed with updating the graph. In the current example, we have one node with one formula $R$ in **now**. The expansion axiom shown in Eq. 3 tells us that $R$ is true either when $Q$ holds in the current instant, or $P$ holds in the current instant and $P\mathbf{U}Q$ holds at the next instant. We encode this by splitting the current node into two:

- In one of the new nodes, $Q$ is added to **now**. This corresponds to the case where $Q$ holds in the current instant.
- In the other new node, $P$ is added to **now**, and $P\mathbf{U}Q$ is added to **next**. This corresponds to the case where $P$ holds in the current instant, and $P\mathbf{U}Q$ in the next.

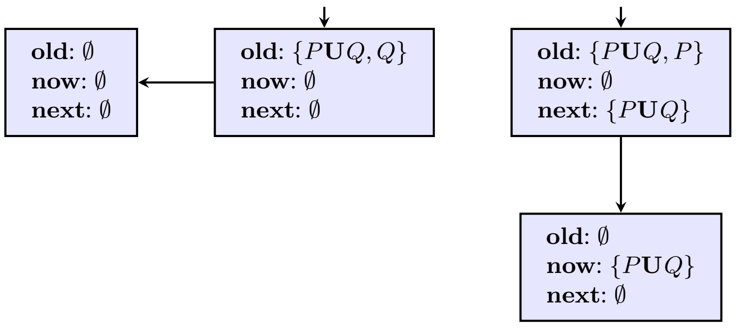
The new state of our graph is:


**old**: $\{P\mathbf{U}Q\}$
**now**: $\{Q\}$
**next**: $\emptyset$
**old**: $\{P\mathbf{U}Q\}$
**now**: $\{P\}$
**next**: $\{P\mathbf{U}Q\}$

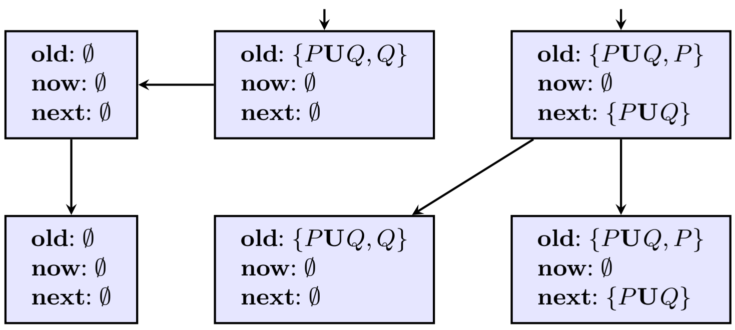
Continuing on, both of the nodes in our graph now contain **now** sets comprised of a single literal. In general, when we process a literal from **now**, we first check to see if its negation is in **old**. If it is, then we discard the current node, because it is a contradiction: no trace can satisfy both $P \wedge \neg P$. If the negation of the literal is not in **old**, then we simply update the graph by removing the literal from **now** and adding it to old. This brings our running example to the following graph.


**old**: $\{P\mathbf{U}Q, Q\}$
**now**: $\emptyset$
**next**: $\emptyset$
**old**: $\{P\mathbf{U}Q, P\}$
**now**: $\emptyset$
**next**: $\{P\mathbf{U}Q\}$

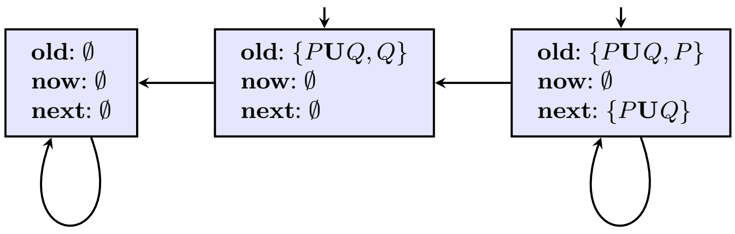
At this point both of the nodes in our graph have empty **now** sets. We proceed by selecting a node for which to create a successor, by setting the **now** set of the successor to the **next** set of the selected node. The **old** and **next** sets of the successor are initialized to be empty. In the following graph, we have taken this step for both nodes.



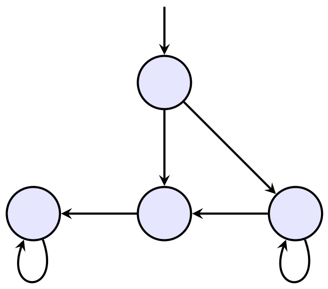
We continue processing nodes with non-empty **now** sets just as we did before. Because the new node with **now** $= \{P\mathbf{U}Q\}$ is exactly the same as the one we started out with, we arrive at the following after multiple steps of processing.


old: $\emptyset$ now: $\emptyset$ next: $\emptyset$
old: $\{P\mathbf{U}Q, Q\}$ now: $\emptyset$ next: $\emptyset$
old: $\{P\mathbf{U}Q, P\}$ now: $\emptyset$ next: $\{P\mathbf{U}Q\}$
old: $\emptyset$ now: $\emptyset$ next: $\emptyset$
old: $\{P\mathbf{U}Q, Q\}$ now: $\emptyset$ next: $\emptyset$
old: $\{P\mathbf{U}Q, P\}$ now: $\emptyset$ next: $\{P\mathbf{U}Q\}$

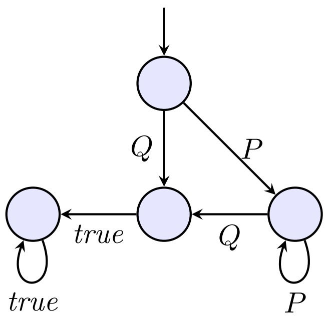
We currently have a graph with nodes that share **old** and **next** sets, and have empty **now** sets. These are redundant, and we remove the more recent copy of each such redundant node by adding its incoming edges to the other's incoming set.


old: $\emptyset$ now: $\emptyset$ next: $\emptyset$
old: $\{P\mathbf{U}Q, Q\}$ now: $\emptyset$ next: $\emptyset$
old: $\{P\mathbf{U}Q, P\}$ now: $\emptyset$ next: $\{P\mathbf{U}Q\}$

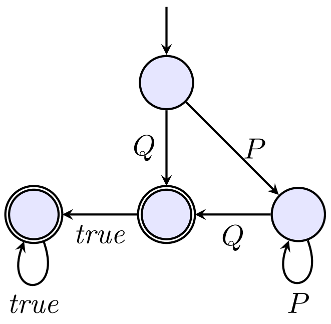
At this point, none of the nodes in our graph have non-empty **now** sets. There is no further processing to be done, so we can finish by constructing a generalized Büchi automaton corresponding to the graph. We clearly want the alphabet to consist of sets of atomic propositions from $\Sigma$. The set of states in the automaton will correspond exactly to the set of states in the graph above, in addition to a new initial state. Likewise, the edges will also remain the same, except with the addition of edges from the new initial state to the states that represent nodes with `init` in their set of incoming edges.



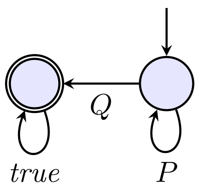
What about the transition labels and accepting states? To assign transition labels, we look to the **old** sets of each node in the graph. The labels on each edge will correspond to the conjunction of all of the atomic propositions in the **old** set of the post-state. The only "corner case" to deal with appears in our example, where one of the node has an empty **old** set. The meaning of such a node is that there are no conditions on the traces reaching the corresponding state; in these cases the transition is labeled with $true$ to reflect this fact. We are left with the following automaton.



Finally, we need to assign the accepting states. Notice that not every infinite path through the automaton belongs in the language of $P\mathbf{U}Q$. In particular, the word $P, P, \ldots$ follows transitions on the automaton, staying in the rightmost state. Indeed, although the node in our graph construction for this state contained $P\mathbf{U}Q$, this word does not traverse a successor state at any point that contains $Q$ as required for inclusion in $\mathcal{L}(P\mathbf{U}Q)$. This is solved by setting as accepting any node for which $P\mathbf{U}Q \notin \mathbf{old}$, or $Q \in \mathbf{old}$. This leaves us with the following automaton.



Note that this is by no means the simplest automaton that accepts the language. It is not hard to see that we can simplify the structure to the following, which is perhaps more natural.



**Other operators** We just saw how to systematically convert the LTL formula $P\mathbf{U}Q$ into a nondeterministic Büchi automaton. In order to generalize the approach to other formulas, we need to consider how to process other operators that appear in the **now**

sets of nodes. When processing a formula consisting of until, we split the current node into two new ones using the expansion axiom for $P\mathbf{U}Q$. We can handle the remaining cases by considering the temporal operator $\circ P$, the Boolean operators $P \wedge Q$, $P \vee Q$, and a new temporal operator $P\mathbf{R}Q$. The release operator is defined to be exactly the dual of until:

$$P\mathbf{R}Q \equiv \neg(\neg P\mathbf{U}\neg Q) \tag{6}$$

The addition of the release operator to LTL formulas allows us to define a useful negation normal form.

**Definition 7** (LTL negation normal form)**.** An LTL formula $P$ is said to be in negation normal form if it contains the operators $\circ$, $\mathbf{U}$, $\mathbf{R}$, $\wedge$, $\vee$, and $\neg$. Additionally, negation in $P$ only occurs over atomic propositions.

**Theorem 8.** *Any LTL formula has an equivalent representation in negation normal form.*

*Proof.* Much as in the case of propositional logic, an LTL formula can be converted to its NNF equivalent with the use of DeMorgan's law and the following equivalences.

$$\Diamond P \leftrightarrow \mathit{true}\mathbf{U}P \tag{7}$$
$$\Box P \leftrightarrow \mathit{false}\mathbf{R}P \tag{8}$$
$$\neg\circ P \leftrightarrow \circ\neg P \tag{9}$$
$$\neg(P\mathbf{U}Q) \leftrightarrow \neg P\mathbf{R}\neg Q \tag{10}$$
$$\neg(P\mathbf{R}Q) \leftrightarrow \neg P\mathbf{U}\neg Q \tag{11}$$

Proving the validity of these equivalences is a good exercise. □

Assuming the formula we wish to convert is in NNF, we can use the following steps to process nodes with $\circ$, $\mathbf{R}$, $\wedge$, and $\vee$ formulas in **now**.

$P \wedge Q$**:** On selecting a conjunction, $P \wedge Q$ is removed from **now** and replaced with $P$ and $Q$.

$P \vee Q$**:** On selecting a disjunction, the current node $q$ is split into $q_1$ and $q_2$. $P$ is added to **now** of $q_1$, and $Q$ to **now** of $q_2$.

$\circ P$**:** On selecting $\circ P$ from **now** in node $q$, simply remove $\circ P$ from **now** and add it to **next**.

$P\mathbf{R}Q$**:** As in the case of $\mathbf{U}$, when processing $P\mathbf{R}Q$ the current node $q$ is split into two new ones $q_1, q_2$. The expansion axiom for release is,

$$P\mathbf{R}Q \leftrightarrow Q \wedge (P \vee \circ(P\mathbf{R}Q)) \leftrightarrow (Q \wedge P) \vee (Q \wedge \circ(P\mathbf{R}Q)) \tag{12}$$

So, $Q$ is added to **now** in both $q_1$ and $q_2$, $P$ is added to **now** in $q_1$, and $P\mathbf{R}Q$ is added to **next** of $q_2$.

Finally, to handle the general case of arbitrary LTL formulas, we need to expand the way in which accepting states are assigned as well. The most natural way to describe the construction constructs a *generalized* Büchi automaton (GBA), which is exactly like a NBA but for the syntax and semantics of the accepting states. Namely, a GBA has potentially multiple distinct sets of accepting states; the acceptance criterion for a word then requires the existence of a run that visits each *set* of accepting states infinitely often.

**Definition 9** (Generalized Büchi Automaton (GBA))**.** 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. $F \subseteq \wp(Q)$ is a set of accepting sets.

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 for each $F_j \in F$, $q_i \in F_j$ 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 $\Sigma$.

While the more complex syntax of GBA make it simpler to describe many algorithms, these automata are no more powerful than NBA. There is a straightforward translation from a GBA to a NBA that accepts the same language. Intuitively, if there are $n$ accepting sets in the GBA, then we make $n$ copies of the automaton each with a single acceptance set. There is a single accepting state in the NBA, which can only be entered when at least one state from all of the original GBA acceptance sets have been traversed.

**General acceptance criteria, complexity** Wrapping up the conversion algorithm, the general criteria for selecting acceptance conditions is as follows. For every subformula of the form $P\mathbf{U}Q$, there is a new acceptance set $F_i \in F$ in the GBA containing the nodes where either $P\mathbf{U}Q \notin$ **old**, or $Q \in$ **old**. This results in an automaton whose size (i.e., number of states) is at most $2^{O(|P|)}$, i.e., exponential in the size of the number of subformulas of the original LTL formula. We will not prove the correctness of this algorithm or its complexity here, but please consult the original paper [GPVW96] for a clearly-written proof of both.

The worst-case exponential time and space requirements may seem dire, and indeed they do pose a problem for some practical applications of the technique. However, two points bear mentioning.

First, the worst-case behavior of the algorithm is oftentimes not encountered in practice. In the example we worked out, there are three subformulas ($\{P, Q, P\mathbf{U}Q\}$, but the resulting automaton only had four states and we saw that simple heuristic optimizations could bring the number of states down to just two. Gerth et al. [GPVW96] showed in their original publication of the approach that many useful commonly-used property formulas yielded small automata well below the worst-case bound. Second, LTL model checking is known to be a hard problem, so it is unlikely that one can do better. See [BKL08] for a proof that the LTL model checking problem is PSPACE-complete.

## References

[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.

[CVWY92] C. Courcoubetis, M. Vardi, P. Wolper, and M. Yannakakis. Memory-efficient algorithms for the verification of temporal properties. *Formal Methods in System Design*, 1(2):275–288, Oct 1992.

[GPVW96] R. Gerth, D. Peled, M. Y. Vardi, and P. Wolper. *Simple On-the-fly Automatic Verification of Linear Temporal Logic*. 1996.

[Tar72] Robert Tarjan. Depth-first search and linear graph algorithms. *SIAM Journal on Computing*, 1(2):146–160, 1972.

[VW86] Moshe Y. Vardi and Pierre Wolper. Automata-theoretic techniques for modal logics of programs. *Journal of Computer and System Sciences*, 32(2):183 – 221, 1986.