15-414: Bug Catching: Automated Program Verification

Lecture Notes on Model Checking LTL with Büchi Automata

Matt Fredrikson

Carnegie Mellon University
Lecture 18
November 4, 2025

Temporal logics give us a precise language for talking about how systems evolve over time. In this course we have seen two closely related perspectives. Computation Tree Logic (CTL) is a branching-time logic: formulas are interpreted at states, and path quantifiers range over the tree of all computations that the system may take from a state. Linear Temporal Logic (LTL) is a linear-time logic: formulas are interpreted over single computations (traces). Both logics let us express safety ("nothing bad ever happens") and liveness ("something good eventually happens") properties, but they differ in how time and nondeterminism are modeled.

Model checking asks the algorithmic question, given a finite-state computation structure (Kripke structure) K and a temporal-logic formula P, does $K \models P$? For CTL we saw a direct solution by computing the semantics by fixpoints over the state space. For LTL this approach is not available because formulas range over traces rather than individual states. Instead, we shift to a language-theoretic viewpoint that will make algorithmic structure apparent.

We ended last time by formulating LTL model checking as a language-inclusion problem. Viewing $\mathcal{L}(K)$ as the set of all traces (behaviors) of K and $\mathcal{L}(P)$ as the set of all traces that satisfy P, correctness of K with respect to P is exactly $\mathcal{L}(K) \subseteq \mathcal{L}(P)$. If these were regular languages over *finite* words, we could decide inclusion by intersecting with a complement and testing emptiness. The obstacle is that both sets are languages of *infinite* words; the familiar DFAs and NFAs are not the right tool.

In this lecture we resolve that mismatch by introducing automata on infinite words, called Büchi automata. With Büchi automata, we recover the basic approach of building a product automata and checking for emptiness. Syntactically, a nondeterministic Büchi automaton (NBA) looks just like an NFA: a finite set of states, an alphabet, a transition function, a set of initial states, and a set of accepting states F. Semantically, the

acceptance condition changes. Instead of ending in F, an accepting run must visit F *infinitely often*. This is what lets NBAs recognize languages of infinite words.

Algorithmically, this last point leads to efficient graph-search procedures. A widely used approach is nested depth-first search (NDFS), which searches for accepting cycles by interleaving two DFS traversals; an equivalent view is to compute strongly connected components and check whether any reachable component contains an accepting state. We will see how these algorithms work in today's lecture.

Learning goals.

- Understand the semantics of nondeterministic Büchi automata and how acceptance over infinite runs differs from NFAs.
- Translate transition structures and simple LTL properties into NBAs, and use closure under intersection via a product construction.
- Reduce LTL model checking to an emptiness check and recognize when cycle detection suffices to decide acceptance.
- See how an efficient nested depth-first search algorithm solves the problem of emptiness checking in Büchi automata.

1 Recap: LTL Semantics and Language View

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 σ obtained by running a computation structure K, $\sigma \models P$.

Definition 1 (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. $\sigma \models \mathbf{X}P$ 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. $\sigma \models \mathbf{U}PQ$ 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, determined by the state at which σ is at any given time.

Definition 2 (LTL semantics (transition systems))**.** Let $K = (W, \curvearrowright, v)$ be a transition system and P an LTL formula over the set of atomic propositions Σ. Then $K \models P$ iff for all traces σ of K, $\sigma \models P$.

Viewing satisfaction as a language now becomes immediate. Let $\mathcal{L}(P)$ be the set of traces that satisfy P, and let $\mathcal{L}(K)$ be the set of traces generated by K. Then correctness is exactly the inclusion

$$\mathcal{L}(K) \subseteq \mathcal{L}(P).$$

Over finite words we could decide inclusion by intersecting with a complement and checking emptiness:

$$\mathcal{L}(K) \subseteq \mathcal{L}(P) \quad \text{iff} \quad \mathcal{L}(K) \cap \overline{\mathcal{L}(P)} = \emptyset.$$

For LTL this plan still makes sense, but it requires an automaton model that accepts infinite words so that the construction mirrors traces. That is exactly what Büchi automata provide.

2 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 3 (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:

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

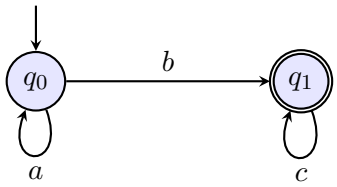
In the above, Σ^ω is the set of all infinite words over alphabet symbols in Σ.

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 3 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.

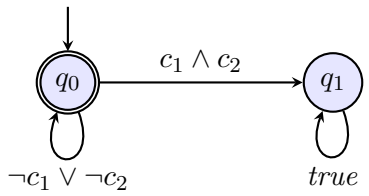


From state q_0, there do not exist any transitions on symbol c. So is the word $acbcccc\ldots$ in the language of this NBA? Looking at the semantics given in Definition 3, we see that it is not. In order to be in the language, there must exist an accepting run, and there is no way to run this NBA on the word $acbcccc\ldots$ because it "falls off" of the transition relation.

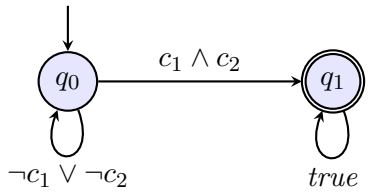
Examples Going back to our original goal of checking the safety and liveness properties of the mutual exclusion example, recall the formula $\Box(\neg c_1 \vee \neg c_2)$. We can represent this property using a NBA, by setting the alphabet Σ to be $\wp(\text{atomic propositions}) = \wp(\{c_1, c_2, n_1,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

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

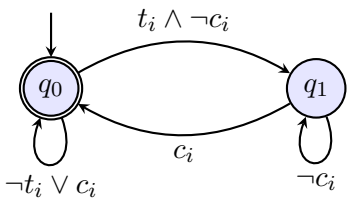
no way to "repair" the trace so that it satisfies the property. The transition diagram is shown below.



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 [Bö62], but we will return to this inconvenience later on.



Looking at another example, let's build an NBA for $\square(t_1 \rightarrow \Diamond c_1) \wedge \square(t_2 \rightarrow \Diamond c_2)$. Because either side of the conjunction is symmetrical with the other, we will show one automaton for $\square(t_i \rightarrow \Diamond c_i)$ 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 4. *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 4 says that a computation structure K is converted to a NBA A_K with the following steps:

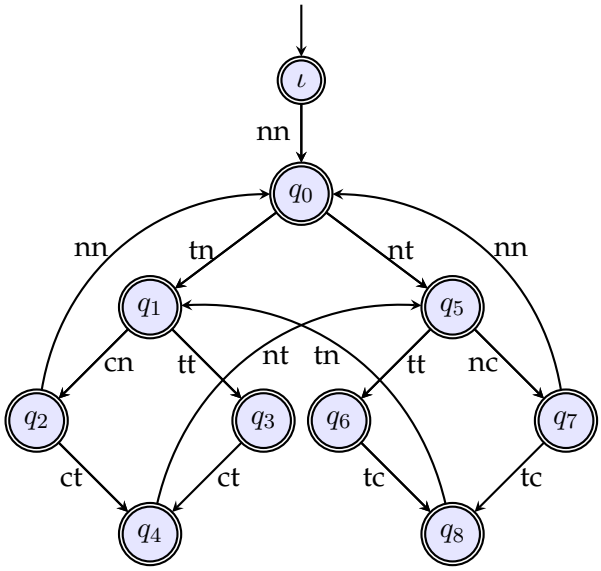
1. The states of A_K are identical to those of K, except a new initial state ι not appearing in K is added. ι is the only initial state of A_K.

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 [CGP99, BKL08].

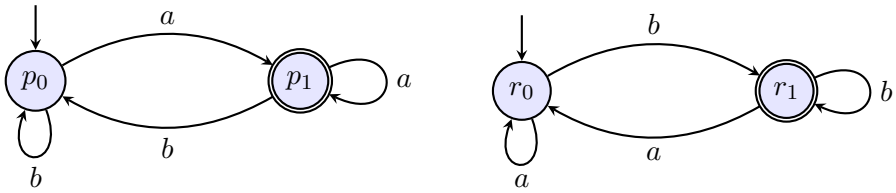
While this construction is straightforward, one does need to be careful about the acceptance set of the product NBA. In particular, when taking the product of $A_1 = (Q_1, \Sigma_1, \delta_1, Q_1^0, F_1)$ and $A_2 = (Q_2, \Sigma_2, \delta_2, Q_2^0, F_2)$, we need to ensure that words accepted by $A_1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

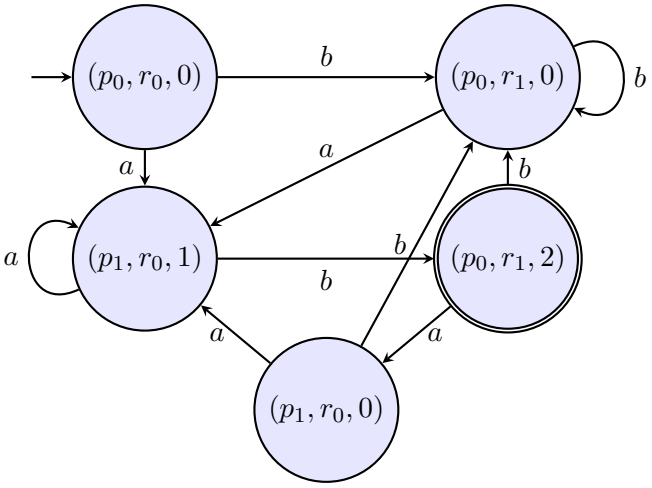
Theorem 5. *Given two nondeterministic Büchi automata* $A_1 = (Q_1, \Sigma, \delta_1, Q_1^0, Q_1)$ *and* $A_2 = (Q_2, \Sigma, \delta_2, Q_2^0, F)$*, the product* $A_{1\cap 2} = (Q_1 \times Q_2, \Sigma, \delta', Q_1^0 \times Q_2^0, Q_1 \times F)$*, where* $(q_1', q_2') \in \delta'((q_1, q_2), \sigma)$ *iff* $(q_i') \in \delta(q_i, \sigma)$ *for* $i = 1, 2$*, satisfies* $\mathcal{L}(A_{1\cap 2}) = \mathcal{L}(A_1) \cap \mathcal{L}(A_2)$*.*

Example. Let $\Sigma = \{a, b\}$. Suppose that A_1 recognizes words with infinitely many a's, and A_2 recognizes words with infinitely many b's. Therefore the product should accept words that have both infinitely many a's and infinitely many b's.



Here $F_1 = \{p_1\}$ and $F_2 = \{r_1\}$. Using the standard three-part (0/1/2) product sketched above, we build states $Q' = Q_1 \times Q_2 \times \{0, 1, 2\}$ with initial copy 0. Intuitively, 0 means "waiting for F_1", 1 means "saw F_1, now waiting for F_2", and 2 is a one-step checkpoint that we make *accepting*. Formally, on a letter $x \in \Sigma$ we take synchronous moves on A_1 and A_2 and update the copy as follows: from copy 0, if the A_1 successor is in F_1 go to copy 1 (else stay in 0); from copy 1, if the A_2 successor is in F_2 go to copy 2 (else stay in 1); from copy 2, go to copy 0 on the next step. The accepting set is $F' = Q_1 \times Q_2 \times \{2\}$.

The picture below shows a reachable fragment of the product, starting at $(p_0, r_0, 0)$. Accepting nodes (copy 2) are double circled. Along the word $(ab)^\omega$ the run cycles through copies $0 \to 1 \to 2 \to 0 \to \cdots$, visiting copy 2 infinitely often, so the product accepts exactly the intersection $\mathsf{Inf}(a) \cap \mathsf{Inf}(b)$.



The accepting copy 2 in the product forces every accepting run to witness F_1 and then F_2 infinitely often; words that are missing infinitely many a's or b's fail to reach copy 2 infinitely many times and are rejected.

3 Emptiness Checking

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

```
procedure nested_dfs()
  outer_stack := []
  visited_outer := {}
  call outer(s0)

procedure outer(s)
  push(s, outer_stack)
  for all t in post(s) do
    if (t not in visited_outer
        and t not in outer_stack) then
          call outer(t)
  if s in accepting then
    inner_stack := []
    call inner(s)
  pop(s, outer_stack)
  visited_outer += {s}
```

```
procedure inner(s)
  for all t in post(s) do
    if t in outer_stack then
      report cycle
    else if t not in inner_stack then
      push(t, inner_stack)
      call inner(t)
      pop(t, inner_stack)
```

Figure 1: Nested depth-first search of Schwoon and Esparza [SE05].

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.

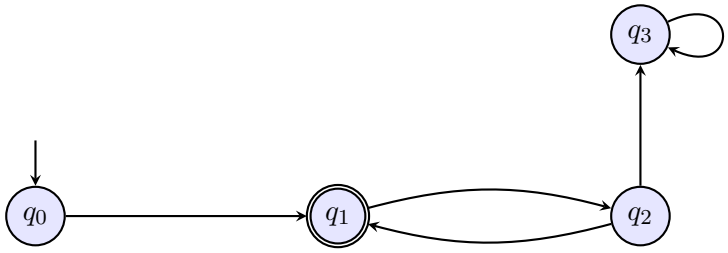
Whenever such a strongly-connected component exists in the NBA, there will necessarily be a cycle from some accepting state back to itself; given a strongly-connected component with an accepting state, it is always possible to find such a cycle, and the converse clearly holds. So given a product automaton 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 one must keep in mind that the size of the LTL automaton can grow double-exponentially through the process of taking its complement.

Nested Depth-First Search. A popular approach for determining the presence of reachable accepting SCCs is the *nested depth-first search* algorithm. Nested depth-first search (NDFS) decides emptiness of an NBA by interleaving two DFS traversals and looking for a reachable accepting cycle. The outer search locates reachable accepting states. When an accepting state is identified reachable, it triggers the inner search to look for a path back to that accepting state. The algorithm is shown in Figure 1.

There are two things to Note in Figure 1. First, the outer search traverses its unvisited successors eagerly, before checking to see if the current state is accepting and looking for a cycle using the inner search. This ensures that the inner search will only traverse states that have already been fully explored by the outer search, and that we only check those states once for cycles. If the outer search used a pre-order, calling the inner search eagerly before recursing, then the full search could end up traversing portions of the state space multiple times, blowing up the complexity.

The second thing to note is that the inner search does not explicitly look for a traversal back to the accepting state from which it began q_a. Instead it checks to see if a successor is already on the outer stack. This is sufficient, because we know that q_a is reachable from all the states on the outer search's stack. Any accepting cycle that goes through q_a must include a back-edge from some reachable state to a state on the stack (which includes q_a). By avoiding re-traversal of states in this way, the overall complexity remains linear $O(|Q| + |\delta|)$.

Example. The NBA below has an accepting cycle that goes through q_1 and q_2, plus a non-accepting branch through q_3.



outer stack	**inner stack**
$[q_0]$	$[\,]$
$[q_0, q_1]$	$[\,]$

Outer takes the first transition to q_1. Before calling the inner search to look for a cycle back to q_1, the outer search will finish exploring q_1's successors.

outer stack	**inner stack**
$[q_0, q_1, q_2, q_3]$	$[\,]$

Outer follows $q_1 \to q_2$ and then tries the side branch to q_3, which has only a self-loop. Since it is not accepting, the search returns to q_2. The only other transition from q_2 also leads back to a state on the outer stack, so we return to q_1.

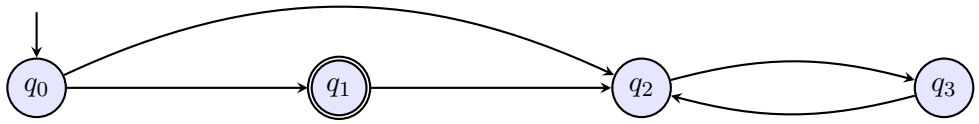
outer stack	**inner stack**
$[q_0, q_1]$	$[]$

Because q_1 is accepting, we launch the inner search from q_1 while q_1 is still on the outer stack. At this point, the inner search scans q_1's successors, q_2, which is not currently on the outer stack. It is added to the inner stack, and the inner search continues from q_2.

outer stack	**inner stack**
$[q_0, q_1]$	$[q_2]$

Now, q_1 is a successor of q_2, and currently on the outer stack, so the cycle is detected.

Example. Here the accepting state is reachable, and there *is* a reachable cycle—but it does not contain the accepting state. NDFS will chase the inner search into the non-accepting cycle and, correctly, fail to close a loop to the outer stack.



outer stack	**inner stack**
$[q_0]$	$[\,]$
$[q_0, q_1]$	$[\,]$

Outer reaches q_1 and keeps q_1 on its stack while exploring successors.

outer stack	**inner stack**
$[q_0, q_1, q_2, q_3]$	$[\,]$

Outer discovers the $q_2 \leftrightarrow q_3$ cycle; neither vertex is accepting, so any cycle confined to $\{q_2, q_3\}$ will not satisfy the acceptance condition.

blue (outer) stack	**red (inner) stack**
$[q_0, q_1]$	$[q_1]$

Before popping accepting q_1, NDFS launches the inner search from q_1. The inner DFS follows $q_1 \to q_2$ into the $q_2 \leftrightarrow q_3$ cycle, but every path stays within $\{q_2, q_3\}$ and never reaches a node currently on the outer stack; the inner search therefore terminates without reporting a cycle.

blue (outer) stack	**red (inner) stack**
$[q_0]$	$[\,]$

Outer pops q_1 and returns to q_0. The remaining successor q_2 is already fully explored by the outer search, so it is skipped. Since no accepting cycle is reachable, NDFS concludes the language is empty. Because the only reachable cycle avoids accepting states, no accepting cycle exists and emptiness holds.

Remarks. In practice, NDFS is run on the product automaton $A_{K \cap \overline{P}}$. A found accepting cycle directly yields a counterexample trace for $K \models P$. When no accepting cycle exists, the product language is empty and we conclude $K \models P$. Many solver implementations use small engineering refinements (e.g., compact bitset marks, partial order reduction), but the core idea remains the same: the outer DFS builds a candidate prefix to an accepting state, and the inner DFS confirms (or refutes) the existence of a cycle through it.

References

[Bö62] 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.

[SE05] Stefan Schwoon and Javier Esparza. A note on on-the-fly verification algorithms. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2005)*, volume 3440 of *Lecture Notes in Computer Science*, pages 174–190. Springer, 2005. doi:10.1007/978-3-540-31980-1_12.