Formal Verification Methods

Lectures by Doron Peled

Summary by Ari Feiglin (ari.feiglin@gmail.com)

Contents

1 Transition Systems

When modelling systems, one must take into consideration a variety of factors: for example, is the system sequential or concurrent? When investigating the transitions between states, how granular should they be? These questions are common questions in computer science, the terms may not be though. A sequential system is a system with only one thread of execution, while a concurrent system may be multi-threaded/multiprogrammed/multiprocessed. The granularity of a transition refers to how detailed we view the transition: is the command `x` := `y` atomic? Or do the variables first need to be loaded into memory?

We now begin to discuss how we model systems.

1.0.1 Definition

A **transition system** over a first-order language $\mathcal{L}$ is a triplet $(\mathcal{S}, T, \Theta)$, where

(1) $\mathcal{S}$ is a (potentially many-sorted) $\mathcal{L}$-structure. The symbols of $\mathcal{L}$ correspond to the symbols utilized within the program in question. For example, $\mathcal{L}$ may contain the $+$ operator, $<$ relation, etc. As opposed to general first-order logic, the set of variables V is taken to be finite here. This set of variables correspond to precisely what you'd expect: the set of all variables in the program. This includes internal registers utilized by the program, called the *program counters*, for which there is one for each concurrent process, and they point to the location of the next instruction to be executed.

(2) T is a *finite* set of **transitions**. Each transition $t \in T$ has the form ($\mathcal{T}_{\mathcal{L}}$ is the set of $\mathcal{L}$-terms)

$$p \longrightarrow (v_1, \dots, v_n) := (e_1, \dots, e_n) \qquad (v_1, \dots, v_n \in V, e_1, \dots, e_n \in \mathcal{T}_{\mathcal{L}})$$

p is a quantifier-free formula in $\mathcal{L}$. Notice that even in concurrent systems, there is a single set of transitions, meaning all the transitions are grouped together.

(3) Θ is the *initial condition*, a quantifier-free formula in $\mathcal{L}$.

In this model, a **state** is an assignment of the variables in V to elements of the domain of $\mathcal{S}$. In other words, a state is a valuation $s\colon V \longrightarrow S$ ($S = dom\mathcal{S}$), so $\mathcal{S}$ together with a state form an $\mathcal{L}$-model. The **state space** is the set of all possible states, which can be taken to be S^V or a subset of this (if for example, $\mathcal{S}$ contains all the naturals, but our computer's memory is bound in size).

A transition of the form $p \longrightarrow (v_1, \dots, v_n) := (e_1, \dots, e_n)$ intuitively can execute from any state which satisfies the condition p. The condition p is called the *enabledness condition* of the transition t, and if p is satisfied by the state s, ie. $\mathcal{S}, s \vDash p$ (recall that $\mathcal{S}, s$ is simply an $\mathcal{L}$-model), then t is said to be *enabled* at s. t transitions from a state s in which it is enabled to a state where the value of each v_i is set to $e_i^{\mathcal{S}}$ for $1 \leq i \leq n$, denoted $s' = t(s) = s[e_1/v_1, \dots, e_n/v_n]$.

Note that the assignment is simultaneous: $(x, y) := (y, x)$ has the effect of swapping the values of x and y. Allowing for simultaneous assignments may seem contrary to the idea of having transitions be atomic. But this again goes back to the notion of granularity: we decide what transitions are atomic, and it can be useful to view assignments, even simultaneous ones, as atomic.

1.0.2 Definition

Given a system $(\mathcal{S}, T, \Theta)$, an **execution** is an infinite sequence of states $s_0, s_1, s_2, \dots$ such that $\mathcal{S}, s_0 \vDash \Theta$ (we will also use the notation $s_0 \vDash^{\mathcal{S}} \Theta$), meaning the first state satisfies the initial condition, and for every $i \geq 0$ one of the following holds:

(1) There exists some transition $p \longrightarrow (v_1, \dots, v_n) := (e_1, \dots, e_n) \in T$ that is enabled at s_i, ie. $s_i \vDash^{\mathcal{S}} p$, and s_{i+1} is obtained by this assignment, meaning $s_{i+1} = s_i[e_1^{\mathcal{S}}/v_1, \dots, e_n^{\mathcal{S}}/v_n]$.

(2) There is no transition enabled at s_i, meaning for every transition $t \in T$ whose enabledness condition is p, $s_i \nvDash^{\mathcal{S}} p$. In this case, for every $j \geq i$ we set $s_j = s_i$. So in such a case, we manually extend the sequence if it can no longer be extended.

Instead of the second condition, we could add a new transition to T of the form $\neg(p_1 \vee \cdots \vee p_n) \rightarrow (v := v)$ where $p_1, \dots, p_n$ exhaust all the enabledness conditions of transitions in T, and $v \in V$ is arbitrary. Alternatively we could allow for finite sequences of states, provided the final state enables no transition.

A state which appears in some execution of a program (system) is called *reachable*. Not every state needs to be reachable: consider a program that can hold (bounded) natural numbers with variables y_1, y_2 and the program is written in such a way that $y_1 \geq y_2$ always. But the state $s[y_1] = 1$ and $s[y_2] = 2$ is a valid, yet unreachable, state.

We can view the execution of a system as a *scheduler* which can generate interleaved sequences (sequences where a single transition is executed at a time)

```
function Scheduler($\mathcal{S}, T, \Theta$)
    choose some initial state $s$ such that $s \vDash^{\mathcal{S}} \Theta$
    while ($s$ has an enabled transition)
        choose a transition $t$ enabled by $s$
        $s \leftarrow t(s)$
    end while
    ▷ Extend the sequence infinitely if the final state has no enabled transition
    repeat $s$ forever
end function
```

This scheduler is non-deterministic as the choice for the initial state and the choices between transitions enabled at each state along the execution are made non-deterministically.

1.0.3 Example

Let us give an example of *mutual exclusion*: we have two programs sharing a shared *critical section* (here the variable `turn`):

```
routine Program1
    while (true)
        ▷ wait until turn is zero
        wait(turn = 0)
        turn ← 1
    end while
end routine
```

```
routine Program2
    while (true)
        ▷ wait until turn is one
        wait(turn = 1)
        turn ← 0
    end while
end routine
```

In this example, we have three variables: `turn`, the first program counter $\texttt{pc}_1$, and the second program counter $\texttt{pc}_2$. The transitions are as follows:

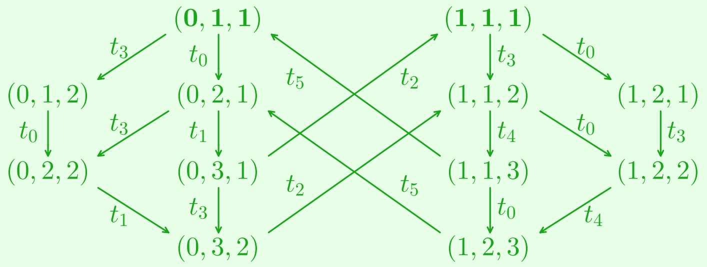
$$t_0\colon \texttt{pc}_1 = 1 \longrightarrow \texttt{pc}_1 := 2,\ t_1\colon (\texttt{pc}_1 = 2 \wedge \texttt{turn} = 0) \longrightarrow \texttt{pc}_1 := 3,\ t_2\colon (\texttt{pc}_1 = 3) \longrightarrow (\texttt{pc}_1, \texttt{turn}) := (1, 1)$$

$$t_3\colon \texttt{pc}_2 = 1 \longrightarrow \texttt{pc}_2 := 2,\ t_4\colon (\texttt{pc}_2 = 2 \wedge \texttt{turn} = 1) \longrightarrow \texttt{pc}_2 := 3,\ t_5\colon (\texttt{pc}_2 = 3) \longrightarrow (\texttt{pc}_2, \texttt{turn}) := (1, 0)$$

Then the initial condition is

$$\Theta = \texttt{pc}_1 = 1 \wedge \texttt{pc}_2 = 1$$

Viewing states as $(\texttt{turn}, \texttt{pc}_1, \texttt{pc}_2)$, then we can draw the following diagram for the transition system, initial states are bold:



Now notice that we do indeed have mutual exclusion, where formally this means always $\neg(\texttt{pc}_1 = 3 \wedge \texttt{pc}_2 = 3)$. Furthermore we have that if $\texttt{turn} = 0$ then eventually $\texttt{turn} = 1$, to prove this we must go through every possible execution which starts with $\texttt{turn} = 0$ and to show that eventually $\texttt{turn} = 1$.

Say instead of implementing `wait` via a lock (eg. mutex), we utilize busy waiting, adding the following two transitions:

$$t_1'\colon (\texttt{pc}_1 = 2 \wedge \texttt{turn} = 1) \longrightarrow \texttt{pc}_1 := 2, \qquad t_4'\colon (\texttt{pc}_2 = 2 \wedge \texttt{turn} = 0) \longrightarrow \texttt{pc}_2 := 2$$

then we no longer have that if $\texttt{turn} = 0$ then eventually $\texttt{turn} = 1$. For example $(0, 1, 1) \rightarrow (0, 1, 2)$ and then $(0, 1, 2)$ is extended forever via t_4'.

In the above example, the focus was more on the states and the possible transitions between them rather than the explicit content of each transition. We can generalize this idea to the concept of *state spaces*:

1.0.4 Definition

A **state space** is a triplet (S, Δ, I), where S is a set of states, $\Delta \subseteq S \times S$ is the transition relations, and $I \subseteq S$ are the initial transitions. This defines a graph, called an **automaton**. A **run** of the automaton is a sequence $s_0 s_1 s_2 \ldots$ such that $s_0 \in I$ is an initial state and for every $i \geq 0$, $(s_i, s_{i+1}) \in \Delta$. Such a run must be maximal, meaning it is either infinite or it reaches a state with no successor.

Sometimes we give names to the transitions in Δ in which case our state space becomes (S, Δ, Σ, I) where Δ now is a subset of $S \times \Sigma \times S$. Every transition gets its own name, so if $(s, \alpha, s'), (r, \alpha, r') \in \Delta$ then $s = r$ and $s' = r'$.

In particular, a transition system defines a state space where S is the set of all states, which are valuations $V \longrightarrow \mathcal{S}$. Then $(s, s') \in \Delta$ if and only if there is a transition t enabled at s such that $s' = t(s)$. And I is the set of states which satisfy the initial condition, $I = \{s \in S \mid s \vDash \Theta\}$.

Suppose we have n concurrent processes, each with a variable v_i and the transitions

$$t^i_1\colon v_i = 1 \longrightarrow v_i := 2, \quad t^i_2\colon v_i = 2 \longrightarrow v_i := 3, \quad t^i_3\colon v_i = 3 \longrightarrow v_i := 1$$

in other words, if v_i is 1, then it is 2, then it is 3, then it is 1. Since this is a concurrent system, we must combine these states together, and then we get that the number of global states becomes 3^n (each state is $(v_1, \ldots, v_n)$ and each v_i can take on three values). This is called *combinatorial explosion*: a relatively simple transition system becomes exponentially larger with the growth of concurrent processes.

Let us examine this above example more closely: notice how we took multiple transition systems and combined them into one. We will define this notion formally: suppose we have a transition system whose transitions T is constructed from *local* components $T_1, \ldots, T_n$. Here, each T_i refers to a local component of the system, be it a concurrent process, a variable, or whatever. For each T_i we also have a set of *transition names* Σ_i and a *labelling function* which is a bijection $L_i\colon T_i \longrightarrow \Sigma_i$. Importantly while the T_is are disjoint, Σ_i need not be.

If two transitions have the same name, then we execute them together. Formally, from each global state s, we can execute all the transitions with the name d (meaning $L_i(t) = d$) provided *all of them* are enabled at s. So suppose we have the transitions $t_i\colon p_i \longrightarrow (v^i_1, \ldots, v^i_{n_i}) := (e^i_1, \ldots, e^i_{n_i})$ for $1 \leq i \leq k$ such that $L_i(t_i) = d$ for all $1 \leq i \leq k$. Then the resulting transition is

$$\big(p_1 \wedge \cdots \wedge p_k\big) \longrightarrow (v^1_1, \ldots, v^1_{n_1}, \ldots, v^k_1, \ldots, v^k_{n_k}) := (e^1_1, \ldots, e^1_{n_1}, \ldots, e^k_1, \ldots, e^k_{n_k})$$

Now suppose that each local component can be represented as a local state space $G_i = (S_i, \Sigma_i, \Delta_i, I_i)$ which corresponds to the local component T_i. We assume that the set of local states S_i are disjoint, but Σ_i need not be. If the label α appears in both Σ_i and Σ_j, then G_i and G_j must be synchronized to perform α at the same time.

We define the operator $\circ$ to combine two state spaces G_1 and G_2 as follows: let $G_1 \circ G_2 = (S, \Sigma, \Delta, I)$ as follows:

(1) $S = S_1 \times S_2$, each state is a pair of a state from G_1 and G_2,

(2) $\Sigma = \Sigma_1 \cup \Sigma_2$, the transition names include all the names in both G_1 and G_2,

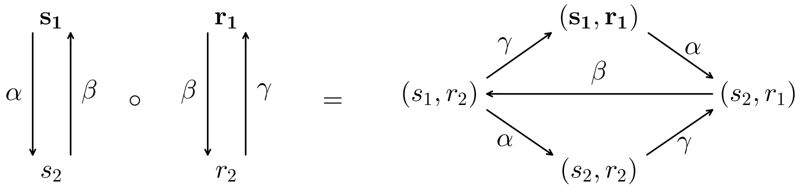
(3) The set of transitions Δ is the union of the following three sets:

  (1) $\big\{\big((s,r), \alpha, (s',r)\big) \bigm| (s, \alpha, s') \in \Delta_1,\ \alpha \in \Sigma_1 \setminus \Sigma_2,\ r \in S_2\big\}$. In this case, we have a transition (s, α, s') in G_1 with no transition of the same name in Σ_2, so we transition from (s, r) to (s', r), leaving the state in G_2 unchanged.

  (2) $\big\{\big((s,r), \beta, (s,r')\big) \bigm| (r, \beta, r') \in \Delta_2,\ \beta \in \Sigma_2 \setminus \Sigma_1,\ s \in S_1\big\}$. This is similar to the previous set, but for G_2 instead of G_1.

  (3) $\big\{\big((s,r), \gamma, (s',r')\big) \bigm| (s, \gamma, s') \in \Delta_1,\ \gamma \in \Sigma_1 \cap \Sigma_2,\ (r, r') \in \Delta_2\big\}$. Here, we have a transition in both G_1 and G_2, so the transition is done simultaneously.

So for example, the following two state spaces combine together to give



2 Specification Formalisms

We now introduce language which allows us to formally discuss properties of systems and their executions. By doing so, we can prove these properties formally and without room for interpretative error.

Let $\mathcal{L}$ be a set logic (either propositional or first-order), $\mathcal{S}$ will be an $\mathcal{L}$-structure, but in general we will refrain from mentioning it instead; we will write $\vDash$ in place of $\vDash^{\mathcal{S}}$.

2.0.1 Definition

Linear temporal logic (abbreviated LTL) is an instance of modal logic. It is defined over $\mathcal{L}$ recursively as follows:

(1) Every formula of $\mathcal{L}$ is also a formula of LTL,

(2) if φ and ψ are LTL formulas, so too are $\neg\varphi, (\varphi \wedge \psi), \bigcirc\varphi, \Diamond\varphi, \Box\varphi, \varphi\mathsf{U}\psi, \varphi\mathsf{V}\psi$.

An LTL formula is interpreted over an infinite sequence of states $\xi = x_0x_1x_2\ldots$. Let us write ξ^k for the suffix $\xi^k := x_kx_{k+1}\ldots$, then we define

(1) if $\varphi \in \mathcal{L}$ then $\xi^k \vDash \varphi$ if $x_k \vDash \varphi$ in $\mathcal{L}$,

(2) $\xi^k \vDash \neg\varphi$ if $\xi^k \nvDash \varphi$,

(3) $\xi^k \vDash \varphi \wedge \psi$ if $\xi^k \vDash \varphi$ and $\xi^k \vDash \psi$,

(4) $\xi^k \vDash \bigcirc\varphi$ if $\xi^{k+1} \vDash \varphi$,

(5) $\xi^k \vDash \Diamond\varphi$ if there is an $i \geq k$ such that $\xi^i \vDash \psi$,

(6) $\xi^k \vDash \Box\varphi$ if $\xi^i \vDash \psi$ for every $i \geq k$,

(7) $\xi^k \vDash \varphi\mathsf{U}\psi$ if there is an $i \geq k$ such that $\xi^i \vDash \psi$ and for all $k \leq j < i$, $\xi^j \vDash \psi$,

(8) $\xi^k \vDash \varphi\mathsf{V}\psi$ if for every $i \geq k$, $\xi^i \vDash \psi$; or for some $i \geq k$, $\xi^i \vDash \varphi$ and for every $k \leq j \leq i$, $\xi^j \vDash \psi$.

Intuitively we can explain the new operators as follows:

(4) $\bigcirc$ is the *nexttime* operator: $\bigcirc\varphi$ holds in the sequence $x_kx_{k+1}\ldots$ if φ holds starting from the next state x_{k+1}. Visually we can view it like so:

	φ																													

(5) $\Diamond$ is the *eventually* operator: $\Diamond\varphi$ holds in the sequence ξ provided there exists a suffix in which φ holds. Visually:

										φ																				

(6) $\Box$ is the *always* operator: $\Box\varphi$ holds in the sequence ξ provided it holds in every suffix of ξ. Visually:

φ	φ	φ	φ	φ	φ	φ	φ	φ	φ	φ	φ	φ	φ	φ	φ	φ	φ	φ	φ	φ	φ	φ	φ	φ	φ	φ	φ	φ	φ	φ

(7) U is the *until* operator: $\varphi\mathsf{U}\psi$ holds in the sequence ξ if ψ holds eventually and φ holds up until then:

φ	φ	φ	φ	φ	φ	φ	φ	φ	φ	φ	φ	φ	φ	φ	φ	φ	φ	φ	φ	ψ										

(8) V is the *release* operator: $\varphi\mathsf{V}\psi$ holds if ψ either holds forever, or up until some point when both φ and ψ hold. The reasoning for the name is that φ "releases" ψ from having to hold for forever.

ψ	ψ	ψ	ψ	ψ	ψ	ψ	ψ	ψ	ψ	ψ	ψ	ψ	ψ	ψ	ψ	ψ	ψ	ψ	ψ	ψ	ψ	ψ	ψ	ψ	ψ	ψ	ψ	ψ	ψ	ψ

Or:

ψ	ψ	ψ	ψ	ψ	ψ	ψ	ψ	ψ	ψ	ψ	ψ	ψ	ψ	ψ	ψ	ψ	ψ	ψ	ψ	ψ φ										

Notice that $\Diamond$ is a special case of U: $\Diamond\varphi \equiv \mathsf{true}\mathsf{U}\varphi$. And $\Box$ is a special case of V: $\Box\varphi \equiv \mathsf{false}\mathsf{V}\varphi$ (since false can never relase φ). $\Box$ and diamond are also related through $\neg\Box\varphi \equiv \Diamond\neg\varphi$.

We can also relate U and V by $\neg(\varphi\mathsf{V}\psi) \equiv (\neg\varphi)\mathsf{U}(\neg\psi)$. We will prove this directly from definition:

$$\xi^k \vDash \varphi\mathsf{V}\psi \iff \big((\forall i \geq k)\xi^i \vDash \psi\big) \lor \big((\exists j \geq k)(\forall k \leq i < j)\xi^i \vDash \psi \land \psi^j \vDash \varphi\big)$$

and so

$$\xi^k \vDash \neg(\varphi\mathsf{V}\psi) \iff \big((\exists i \geq k)\xi^i \vDash \neg\psi\big) \land \big((\forall j \geq k)(\exists k \leq i < j)\xi^i \vDash \neg\psi \lor \psi^j \vDash \neg\varphi\big)$$

So at every $j \geq k$, either $\neg\varphi$ or $\neg\psi$ holds, and eventually $\neg\psi$ holds. This just means that $\neg\varphi$ holds until $\neg\psi$ holds, ie. $(\neg\varphi)\mathsf{U}(\neg\psi)$.

Thus, we could've defined LTL with only the operators $\neg, \land, \bigcirc, \mathsf{U}$ (in other words, these form a *complete bundle*).

We can combine operators: for example $\Box\Diamond\varphi$ means that always, φ eventually happens; or equivalently φ happens infinitely many times. $\Diamond\Box\varphi$ means that at some point, φ will hold forever. $\bigcirc\bigcirc\varphi$ means that φ holds after two steps. Notice that $\xi \vDash \Diamond\varphi$ if and only if there exists some n such that $\xi \vDash \bigcirc^n\varphi$ ($\bigcirc^n$ meaning $\bigcirc \cdots \bigcirc$ n times).

Let P be a system which has multiple executions, then we write $P \vDash \varphi$ if $\xi \vDash \varphi$ for all executions ξ of P. Importantlu $P \nvDash \varphi$ does not imply $P \vDash \neg\varphi$, since one execution not satisfying φ does not mean all executions don't satisfy φ.

2.0.2 Example

Let us consider a simple model of a spring. The spring can be in one of the following three states: $\{\mathit{initial}, \mathit{extended}, \mathit{extended\ and\ malfunctioned}\}$ which we denote s_1, s_2, s_3 respectively. So our propositional variables are $PV = \{\mathit{extended}, \mathit{malfunctioned}\}$. Since s_1 is neither extended nor malfunctioned, $s_1 \vDash \neg\mathit{extended} \land \neg\mathit{malfunctioned}$, $s_2 \vDash \mathit{extended} \land \neg\mathit{malfunctioned}$, $s_3 \vDash \mathit{extended} \land \mathit{malfunctioned}$.

We can transition from s_1 to s_2 via pulling the spring, and releasing the spring can either transition to s_1 or to s_3. From s_3 we transition only to s_3.

This system has an infinite number of executions, for example

$$\begin{aligned}\xi_0 &= s_1 s_2 s_1 s_2 s_3 s_3 s_3 s_3 \cdots\\ \xi_1 &= s_1 s_2 s_3 s_3 s_3 s_3 s_3 s_3 \cdots\\ \xi_2 &= s_1 s_2 s_1 s_2 s_1 s_2 s_1 s_2 \cdots\end{aligned}$$

Let us investigate ξ_0:

(1) $\xi_0 \nvDash \mathit{extended}$ since $\mathit{extended}$ is a formula of the underlying logic of the LTL and so ξ_0 satisfies $\mathit{extended}$ if and only if its first state, s_1, does. It does not.

(2) $\xi_0 \vDash \bigcirc\mathit{extended}$ ("nexttime extended") since $\xi_0 \vDash \bigcirc\mathit{extended} \iff \xi_0^1 \vDash \mathit{extended} \iff s_2 \vDash \mathit{extended}$ which it does.

(3) $\xi_0 \nvDash \bigcirc\bigcirc\,\mathit{extended}$ ("nexttime nexttime extended") since ξ_0^2 begins with s_1 which does not satisfy $\mathit{extended}$.

(4) $\xi_0 \vDash \Diamond\mathit{extended}$ ("eventually extended") since eventually the spring is extended (this is since $\xi_0 \vDash \bigcirc\mathit{extended}$).

(5) $\xi_0 \nvDash \Box\mathit{extended}$ ("always extended") since the spring is not always extended.

(6) $\xi_0 \vDash \Diamond\Box\mathit{extended}$ ("eventually always extended") since eventually the spring remains in s_3 where it is extended.

(7) $\xi_0 \nvDash (\neg\mathit{extended})\mathsf{U}\mathit{malfunctioned}$ ("not extended until malfunctioned") since the spring is not extended, then extended and not malfunctioned.

Let us now investigate the system P as a whole:

(1) $P \vDash \Diamond\mathit{extended}$ since for the spring to not extend, it would need to forever remain in s_1, which is impossible.

(2) $P \vDash \Box(\neg\mathit{extended} \to \bigcirc\mathit{extended})$ which means that always, if the spring is not extended then the next time it is. This is since in order for the spring to not be extended, it must be in s_1, which means that the next time it is in s_2, extended.

(3) $P \not\models \Diamond\Box \mathit{extended}$, since ξ_2 is a counterexample: here we have that we never are only extended, in other words $\xi_2 \models \Box\Diamond\neg \mathit{extended}$.

(4) $P \not\models \neg\Diamond\Box \mathit{extended}$, since ξ_0 is a counterexample: here we have that eventually we are only extended.

(5) $P \not\models \Box(\mathit{extended} \to \bigcirc\neg \mathit{extended})$ since it is possible to go from extended to extended (s_2 to s_3). The only sequence in which this is true is ξ_2.

We can form a Hilbert calculus to axiomatize LTL with respect to a system P. To form it we adjoin to the Hilbert calculus of $\mathcal{L}$ (which is either first-order or propositional, usually propositional. But importantly these axioms now range over all LTL formulas, not just formulas in $\mathcal{L}$) the following eight axioms:

(A1) $\neg\Diamond\varphi \leftrightarrow \Box\neg\varphi$ (A2) $\Box(\varphi \to \psi) \to (\Box\varphi \to \Box\psi)$ (A3) $\Box\varphi \to (\varphi \wedge \bigcirc\Box\varphi)$
(A4) $\bigcirc\neg\varphi \leftrightarrow \neg\bigcirc\varphi$ (A5) $\bigcirc(\varphi \to \psi) \to (\bigcirc\varphi \to \bigcirc\psi)$ (A6) $\Box(\varphi \to \bigcirc\varphi) \to (\varphi \to \Box\varphi)$
(A7) $(\varphi \mathsf{U} \psi) \leftrightarrow \big(\psi \vee (\varphi \wedge \bigcirc(\varphi \mathsf{U} \psi))\big)$ (A8) $(\varphi \mathsf{U} \psi) \to \Diamond\psi$

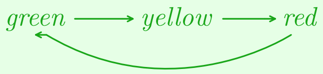
Here we take V as defined by $(\varphi \mathsf{V} \psi) := \neg((\neg\varphi)\mathsf{V}(\neg\psi))$. We use the following rule of reference (temporal generalization) as well as MP

$$\frac{\varphi}{\Box\varphi}$$

meaning that if φ then $\Box\varphi$ (notice that here we do not have an initial state, and hence we obtain the soundness of generalization).

2.0.3 Example

Let us look at a model of a traffic light, which can transition between colors as follows:



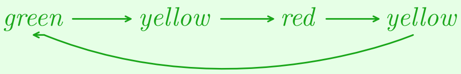
The traffic light is only ever in one color, which can be expressed in LTL as:

$$\Box\big(\neg(\mathit{green} \wedge \mathit{yellow}) \wedge \neg(\mathit{yellow} \wedge \mathit{red}) \wedge \neg(\mathit{red} \wedge \mathit{green}) \wedge (\mathit{green} \vee \mathit{yellow} \vee \mathit{red})\big)$$

Specifying the transition of colors can be done via

$$\Box\big((\mathit{green} \mathsf{U} \mathit{yellow}) \vee (\mathit{yellow} \mathsf{U} \mathit{red}) \vee (\mathit{red} \mathsf{U} \mathit{green})\big)$$

Now suppose we alter the state graph to be



Specifying the colors is now harder, since $\Box(\mathit{yellow} \to \mathit{yellow} \mathsf{U} \mathit{red})$ is no longer necessarily true. We could attemp $\Box\big(((\mathit{green} \vee \mathit{red})\mathsf{U}\mathit{yellow}) \vee (\mathit{yellow}\mathsf{U}(\mathit{green} \vee \mathit{red}))\big)$, but $(\mathit{green} \vee \mathit{red})\mathsf{U}\mathit{yellow}$ allows the light to switch between *green* and *red* before turning *yellow*. A correct specification would be

$$\begin{aligned} \Box\big(\; & (\mathit{green} \to (\mathit{green}\mathsf{U}(\mathit{yellow} \wedge (\mathit{yellow}\mathsf{U}\mathit{red})))) \\ \wedge \; & (\mathit{red} \to (\mathit{red}\mathsf{U}(\mathit{yellow} \wedge (\mathit{yellow}\mathsf{U}\mathit{green})))) \\ \wedge \; & (\mathit{yellow} \to (\mathit{yellow}\mathsf{U}(\mathit{red}\mathsf{U}\mathit{green})))\big) \end{aligned}$$

The first line allows *green* → *yellow* → *red*, the second allows *red* → *yellow* → *green*. These two lines deal only with the case that the light begins on *green* or *red*, the third line deals with the case when it starts on *yellow*.

2.1 Automata on Infinite Words

A *Büchi Automata* is a tuple $\mathcal{A} = (\Sigma, S, \Delta, I, L, F)$ where

(1) Σ is the finite alphabet,

(2) S is a finite set of states,

(3) $\Delta \subseteq S \times S$ is the transition relation,

(4) $I \subseteq S$ is the set of initial states,

(5) $L: S \longrightarrow \Sigma$ is a labeling of the states,

(6) $F \subseteq S$ is the set of accepting states.

A *run* of $\mathcal{A}$ on a word $v \in \Sigma^\omega$ (the set of all infinite words over Σ) is a sequence $\rho: \mathbb{N} \longrightarrow S$ such that

(1) $\rho(0) \in I$, the first state is an initial state,

(2) for all $i \geq 0$, $(\rho(i), \rho(i+1)) \in \Delta$, meaning the transition from $\rho(i)$ to $\rho(i+1)$ is a transition recognized by Δ,

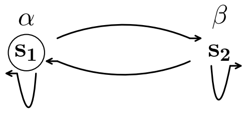
(3) for all $i \geq 0$, $v(i) = L(\rho(i))$, meaning the ith state has the same label as the ith letter in v.

Let us define

$$\inf(\rho) := \{s \in S \mid \text{there exist infinitely many } i \text{ such that } \rho(i) = s\}$$

A run ρ of $\mathcal{A}$ is *accepting* if an accepting state appears infinitely many times in ρ, meaning $\inf(\rho) \cap F \neq \varnothing$. If ρ is a run of the word v, then we say that v is *accepted* by $\mathcal{A}$. The language of $\mathcal{A}$, denoted $\mathcal{L}(\mathcal{A}) \subseteq \Sigma^\omega$, is the set of all words accepted by $\mathcal{A}$.

For the visual representation of Büchi automata, we circle accepting states and bold initial states. So for example



Here the initial states are s_1 and s_2, and the single accepting state is s_1. $\mathcal{L}(\mathcal{A})$ contains letters with infinitely many αs, or as a regular expression $(\beta^*\alpha)^\omega$. This describes an infinite concatenation of words of the form $\beta^*\alpha$ which is formed by an arbitrary number of βs and then an α.

We can alter the definition of a Büchi automaton for a transition system $\mathcal{A}$ as follows:

(1) Σ becomes the set of states of $\mathcal{A}$ (which are valuations of $\mathcal{A}$, not to be confused with the states in S),

(2) L now becomes a labeling function from $S \rightarrow \mathcal{L}$ the set of (propositional) formulas over $\mathcal{A}$,

(3) A run φ of v now must satisfy instead of $v(i) \in L(\rho(i))$ instead that $v(i) \vDash L(\rho(i))$.

We will show that a LTL specification φ can be represented as a Büchi automata $\mathcal{B}$. Then a state space $\mathcal{A}$ satisfies the specification if $\mathcal{L}(\mathcal{A}) \subseteq \mathcal{L}(\mathcal{B})$, meaning every sequence of states valid in $\mathcal{A}$ are valid in $\mathcal{B}$ (meaning they satisfy φ).

3 Automatic Verification

Suppose we have a finite state (transition) system with a set of initial set of states I. We below define an algorithm which searches the state space for every state reachable from the set of initial states.

```
routine SEARCH
let $new$ contain the set of initial states $I$, and $old$ be empty
while ($new$ is not empty)
    choose $s$ from $new$
    remove $s$ from $new$
    add $s$ to $old$
    for ($t$ transition enabled at $s$)
        $s' \leftarrow t(s)$
        if ($s'$ is not in $new$ or $old$) add $s'$ to $new$
    end for
end while
```

Here we do not specify how to choose s from *new* or how to store elements in *new*. One could use a queue for *new*, forming the breadth-first-search (BFS) search algorithm. Or one could use a stack, defining depth-first-search (DFS).

We can add a conditional check to this algorithm to check that every state added to *new* satisfies some property φ (a propositional or first order formula). In this way we can check whether or not φ is an invariant: meaning $\Box\varphi$ holds. Similarly we can check for deadlocks by checking if we visit a state with no successors.

Notice though that this algorithm works only for finite state systems, as there needs to be a finite number of states in order for the algorithm to check every one. So this algorithm cannot work for programs which utilize unbounded integers for example, and larger programs are prone to combinatorial explosion.

3.1 Closure of Büchi Automata

Suppose we have Büchi automata over the same alphabet $\mathcal{A}_i = (\Sigma, S_i, \Delta_i, I_i, L_i, F_i)$ for $i = 1, 2$. We want to define a new Büchi automaton $\mathcal{A} = (\Sigma, S, \Delta, I, L, F)$ such that $\mathcal{L}(\mathcal{A}) = \mathcal{L}(\mathcal{A}_1) \cup \mathcal{L}(\mathcal{A}_2)$. This can be done with relative ease, as we can assume without loss of generality that S_1 and S_2 are disjoint, so define

$$S = S_1 \uplus S_2, \quad \Delta = \Delta_1 \uplus \Delta_2, \quad I = I_1 \uplus I_2, \quad L = L_1 \uplus L_2, \quad F = F_1 \uplus F_2$$

where $L = L_1 \uplus L_2$ means $L(r) = L_1(r)$ for $r \in S_1$ and $L_2(r)$ for $r \in S_2$. Then a run of $\mathcal{A}$ begins with either I_1 or I_2, and then proceeds as it would with $\mathcal{A}_1$ or $\mathcal{A}_2$ respectively.

To define $\mathcal{A}$ for $\mathcal{L}(\mathcal{A}) = \mathcal{L}(\mathcal{A}_\infty) \cap \mathcal{L}(\mathcal{A}_\in)$, we must somehow run both automata in parallel. We attempt a definition as follows:

(1) $S = S_1 \times S_2$,

(2) $((r_1, r_2), (s_1, s_2)) \in \Delta$ if and only if $(r_1, s_1) \in \Delta_1$ and $(r_2, s_2) \in \Delta_2$,

(3) $I = I_1 \times I_2$

(4) $L(r, s) = L_1(r) \wedge L_2(s)$ as these are formulas, if $L(r, s) \equiv \mathsf{false}$ then we can remove (r, s) and all its outgoing and ingoing edges,

Now all that remains is to define F, the set of accepting states. Here we must be careful: as a first attempt we may define $F = F_1 \times F_2$, but then suppose $\mathcal{A}_1$ and $\mathcal{A}_2$ are never simultaneously on an accepting state. Then even though they may both accept the word, since they are never on an accepting state at the same time, this automaton will never be in F. So maybe we try $F = (F_1 \times S_2) \cup (S_1 \times F_2)$? But then if $\mathcal{A}_1$ accepts the word and $\mathcal{A}_2$ doesn't, the automaton would still accept the word.

So let us first define what a *generalized Büchi automaton* is to make this proof easier.

3.1.1 Definition

A **generalized Büchi automaton** is a tuple $\mathcal{A} = (\Sigma, S, \Delta, I, L, F)$ where all the components are the same as for normal Büchi automaton except for F. F is now a set $\{f_1, \ldots, f_m\}$ for $m \geq 0$ where $f_i \subseteq S$. A run ρ is accepted $\mathcal{A}$ if and only if $\inf(\rho) \cap f_i \neq \varnothing$ for all $f_i \in F$.

Now if $\mathcal{A}_1$ and $\mathcal{A}_2$ are two Büchi automaton, we can define their intersection to be a generalized Büchi automaton whose components are defined as above and $F = \{F_1 \times S_2, S_1 \times F_1\}$, so an accepted run must pass through both F_1 and F_2 an infinite number of times.

We now demonstrate how to convert a generalized Büchi automaton into a normal one. Suppose $\mathcal{A} = (\Sigma, S, \Delta, I, L, F)$ is a generalized Büchi automaton with $F = \{f_1, \dots, f_m\}$, then define $S' = \bigcup_{i=1}^m S_i$ where $S_i = S \times \{i\}$ so that all S_i are disjoint. $S' = \bigcup_{i=1}^m S_i$ where $S_i = S \times \{i\}$ so that all S_i are disjoint. We define $\Delta' \subseteq S' \times S'$ as follows:

(1) for $(r, s) \in \Delta$, add $\big((r, i), (s, i)\big)$ to Δ' for all $1 \leq i \leq m$ for which $r \notin f_i$.

(2) if $r \in f_i$ then add $\big((r, i), (s, i+1)\big)$ to Δ' where addition is cyclic: $m + 1 = 1$.

Define $I' = I \times \{1\}$, and $L'(s, i) = L(s)$. Then choose $1 \leq i \leq m$ arbitrarily and set $F' = f_i \times \{i\}$.

To get to a set in f_i, one must first progress through a sequence of sets in $f_1, \dots, f_{i-1}$ since the only way to go between levels (S_is) in S is to reach an accepting state in f_i. Then once one reaches f_i, one must go to the next level S_{i+1}, so to get to f_i again one must progress through $f_{i+1}, \dots, f_m$ and back to $f_1, \dots, f_i$. So to visit f_i an infinite number of times, a run must visit all f_j an infinite number of times.

So a complete construction of $\mathcal{A}_1 \cap \mathcal{A}_2$ is as $(\Sigma, S_1 \times S_2 \times \{1, 2\}, \Delta, I, L, F)$ where

(1) for $(r_1, s_1) \in \Delta_1$ and $(r_2, s_2) \in \Delta_2$ if $r_1 \notin F_1$ and $r_2 \notin F_2$ then $\big((r_1, r_2, i), (s_1, s_2, i)\big) \in \Delta$ for $i = 1, 2$,

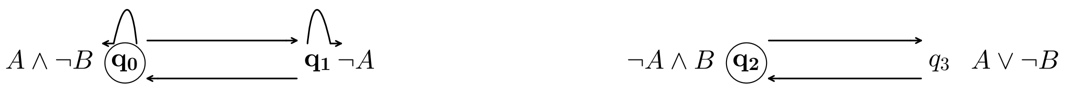
(2) if $r_1 \in F_1$ and $r_2 \notin F_2$ then $\big((r_1, r_2, 1), (s_1, s_2, 2)\big), \big((r_1, r_2, 2), (s_1, s_2, 2)\big) \in \Delta$

(3) if $r_1 \notin F_1$ and $r_2 \in F_2$ then $\big((r_1, r_2, 1), (s_1, s_2, 1)\big), \big((r_1, r_2, 2), (s_1, s_2, 1)\big) \in \Delta$

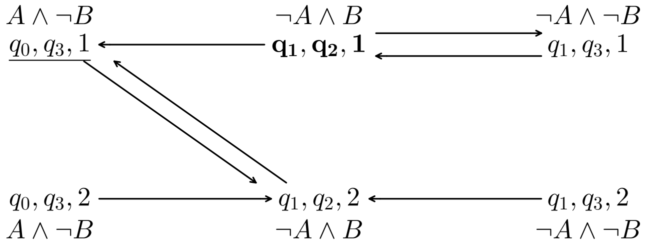
(4) if $r_1 \in F_1$ and $r_2 \in F_2$ then $\big((r_1, r_2, 1), (s_1, s_2, 2)\big), \big((r_1, r_2, 2), (s_1, s_2, 1)\big) \in \Delta$

and $L(r, s, i) = L_1(r) \wedge L_2(s)$ where we remove states if this is equivalent to false. And $I = I_1 \times I_2 \times \{1\}$, $F = F_1 \times S_2 \times \{1\}$.

So for example, if we have



Notice that (q_0, q_2) can be removed as the conjunction of their labels is false. The intersection then is



A more complicated construction is the proof that the complement of a Büchi automaton is also a Büchi automaton.

Recall that to check if $\mathcal{B}$ is a specification for $\mathcal{A}$, we need that $\mathcal{L}(\mathcal{A}) \cap \mathcal{L}(\mathcal{B})^c$ is empty. How do we check if an automaton's language is empty, meaning it has no accepting runs? Suppose $\mathcal{A} = (\Sigma, S, \Delta, I, L, F)$ is a Büchi automaton and ρ is an accepting run of $\mathcal{A}$. Then ρ contains infinitely many states in F, and since S is finite there must exist a suffix ρ' where every state in it occurs infinitely often. This means the states of ρ' comprise of a strongly connected component in the graph (S, Δ): all states are reachable from all the other states in ρ'. Conversely, a strongly connected component reachable from an initial state and which contains an accepting state generates an accepting run: take ρ to first consist of a path from an initial state to the strongly connected component at vertex v, then suppose $a \in F$ is also in the strongly connected component, so v and a are connected, so have the rest of ρ comprise of the path v to a composed with the path from a to v.

So $\mathcal{L}(\mathcal{A})$ being nonempty is equivalent to it having a strongly connected component reachable from an initial state containing an accepting state. So to find if φ is a specification of $\mathcal{A}$, we can utilize the following algorithm:

(1) construct $\overline{\mathcal{B}}$ representing the negation of the specification φ,

(2) construct the intersection $\mathcal{C} = \mathcal{A} \cap \overline{\mathcal{B}}$,

(3) use Tarjan's algorithm to find strongly connected components of $\mathcal{C}$ reachable from initial states,

(4) if none of the components contain an accepting state, then φ is a specification.

(5) otherwise let S be a reachable strongly connected component. Construct a path σ_1 from an initial state to some state accepting state q in the component, and construct a cycle σ_2 from q back to itself. This exists since S is strongly connected, and so $\sigma_1\sigma_2^\omega$ is a counterexample (it satisfies $\mathcal{A}$ but not $\mathcal{B}/\varphi$).

3.2 Translating LTL Formulas into Automata

We will now provide an algorithm for translating propositional LTL formulas into equivalent automata. Suppose φ is an LTL formula, then it is equivalent to a *negation normal form* where negation is applied only on propositional variables. This is due to the following equivalences:

$$\neg\bigcirc\mu \equiv \bigcirc\neg\mu, \quad \neg(\mu\vee\eta) \equiv \neg\mu\wedge\neg\eta, \quad \neg(\mu\wedge\eta) \equiv \neg\mu\vee\neg\eta, \quad \neg\neg\mu \equiv \mu, \quad \neg(\mu\mathsf{U}\eta) \equiv \neg\mu\mathsf{V}\neg\eta, \quad \neg(\mu\mathsf{V}\eta) \equiv \neg\mu\mathsf{U}\neg\eta$$

We also use the equivalences $\Diamond\mu \equiv \mathsf{true}\mathsf{U}\mu$ and $\Box\mu \equiv \mathsf{false}\mathsf{V}\mu$.

The idea is to decompose the formula into a boolean structure, then split the formula into what has to be true in this state and what has to be true in the next state onward. For example, $\varphi\mathsf{U}\psi$ is equivalent to $\psi \vee (\varphi \wedge \bigcirc(\varphi\mathsf{U}\psi))$, so either ψ holds now, or φ holds now and $\varphi\mathsf{U}\psi$ holds from the next state. Similarly $\varphi\mathsf{V}\psi$ is equivalent to $(\varphi\wedge\psi) \vee (\psi \wedge \bigcirc(\varphi\mathsf{V}\psi))$, so either both φ and ψ hold now, or ψ holds now and $\varphi\mathsf{V}\psi$ holds from the next state. So let us define the functions $\mathsf{now}_1, \mathsf{next}, \mathsf{now}_2$ to encode this information, defined as in the table below:

η	$\mathsf{now}_1(\eta)$	$\mathsf{next}_1(\eta)$	$\mathsf{now}_2(\eta)$
$\varphi\mathsf{U}\psi$	$\{\varphi\}$	$\{\varphi\mathsf{U}\psi\}$	$\{\psi\}$
$\varphi\mathsf{V}\psi$	$\{\psi\}$	$\{\varphi\mathsf{V}\psi\}$	$\{\varphi,\psi\}$
$\varphi\vee\psi$	$\{\varphi\}$	$\varnothing$	$\{\psi\}$
$\varphi\wedge\psi$	$\{\varphi,\psi\}$	$\varnothing$	$-$
$\bigcirc\varphi$	$\varnothing$	$\{\varphi\}$	$-$

The meaning being that for η to hold, either everything in $\mathsf{now}_1(\eta)$ holds in this state and $\mathsf{next}_1(\eta)$ holds in the next state, or $\mathsf{now}_2(\eta)$ holds in this state. When the set is $\varnothing$ then it holds vaccuously, and if the set is $-$ then it does not hold.

Our algorithm will utilize *graph nodes* to represent LTL formulas. Each graph node will have the following fields:

(1) *Name*: a unique identifier for the node,

(2) *Incoming*: a list of identifiers (names) for nodes whith edges outgoing into the graph node,

(3) *Now, Old, Next*: each of these represent LTL formulas. *Now* represents the LTL formula which must be satisfied by the current state, *Old* represents the LTL formula which must've been satisfied by the previous state, and *Next* represents the formula satisfied by the next state.

When translating an LTL formula in negation normal form φ, the algorithm begins with a single graph node init with fields $\mathsf{now} = \{\varphi\}$, $\mathsf{next} = \mathsf{old} = \varnothing$. When recursing on the current node s, the algorithm checks if there are any formulas in the now field. If not, then the processing on the current node is complete and now the node must be added to a set of nodes *Node_Set*. If there already exists a node r in *Node_Set* with the same next and old fields (since now is empty), then there is no need to add s to the set. Instead add the incoming edges of s to the incoming edges of r.

Otherwise, add s to *Node_set* and create a *successor node* s' defined so that s''s now field is s's next field. s''s old and next are set to be empty. Now recurse on s'.

If s's now field is non-empty, select a formula η. η is either a proposition, a boolean constant, or the negation of a proposition, or of the form $\neg\mu$, $\mu\vee\psi$, $\mu\wedge\psi$, $\bigcirc\mu$, $\mu\mathsf{U}\psi$, or $\mu\mathsf{V}\psi$. So we split into cases:

(1) η is a proposition, boolean constant, or negation of a proposition. If η is false or $\neg\eta$ is in old then discard s: it contains a contradiction. Otherwise have s *evolve* into s' which is obtained by moving η from now to old.

(2) If $\eta = \mu\mathsf{U}\psi$ then *split* s into two new nodes s_1 and s_2. For s_1, μ is added to now and $\mu\mathsf{U}\psi$ to next. For s_2, ψ is added to now. s_1 and s_2 get the incoming nodes of s. This is due to the fact that $\mu\mathsf{U}\psi$ is equivalent to $\psi \vee (\mu \wedge \bigcirc(\mu\mathsf{U}\psi))$, so for $\mu\mathsf{U}\psi$ we can either go to ψ (s_2) or μ with the next state $\mu\mathsf{U}\psi$ (s_1).

(3) If $\eta = \mu\mathsf{V}\psi$ then *split* s into s_1 and s_2 where $\mathsf{now}(s_1) = \{\psi,\mu\}$, $\mathsf{now}(s_2) = \{\psi\}$, and $\mathsf{next}(s_2) = \{\mu\mathsf{V}\psi\}$.

(**4**) If $\eta = \mu \vee \psi$ then split to s_1, s_2 where $\mathsf{now}(s_1) = \{\mu\}$, $\mathsf{now}(s_2) = \{\psi\}$.

(**5**) If $\eta = \mu \wedge \psi$ then evolve to s' where $\mathsf{now}(s') = \{\mu, \psi\}$.

(**6**) If $\eta = \bigcirc\mu$ then evolve to s' where $\mathsf{next}(s') = \{\mu\}$.

The algorithm then recurses on s'. Once finished recursing, the algorithm has constructed a set of graph nodes *Node_Set*, which it will use to construct a generalized Büchi automaton $B = (\Sigma, S, \Delta, I, L, F)$ defined by

(**1**) Σ is the alphabet which consists of all Boolean combinations of propositional variables found in φ.

(**2**) S is the set of states consisting of nodes in *Node_Set*.

(**3**) $(s, s') \in \Delta$ if and only if s is in $\mathsf{incoming}(s)$.

(**4**) $s \in I$ if and only if $\mathsf{init} \in \mathsf{incoming}(s)$.

(**5**) $L(s)$ is the conjunction of the literals (negated and non-negated propositions) in $\mathsf{old}(s)$.

(**6**) For every subformula of φ of the form $\mu\mathsf{U}\psi$ form a set $f \in F$ which contains all the states s such tha $\psi \in \mathsf{old}(s)$ or $\mu\mathsf{U}\psi \notin \mathsf{old}(s)$.

```
1.  function EXPAND($s$, $Node\_Set$)
2.      if ($\mathsf{now}(s) = \varnothing$)
3.          if ($\exists r \in Node\_Set$ such that $\mathsf{old}(r) = \mathsf{old}(s)$ and $\mathsf{next}(r) = \mathsf{next}(s)$)
4.              $\mathsf{incoming}(r) \leftarrow \mathsf{incoming}(r) \cup \mathsf{incoming}(s)$
5.              return $Node\_Set$
6.          else
7.              $Node\_Set \leftarrow Node\_Set \cup \{s\}$
8.              $\mathsf{now}(s'), \mathsf{next}(s') \leftarrow \varnothing$
9.              $\mathsf{old}(s') \leftarrow \mathsf{now}(s)$
10.             $\mathsf{incoming}(s') \leftarrow \{s\}$
11.             return EXPAND($s, Node\_Set$)
12.         end if
13.     end if
14.     choose $\eta \in \mathsf{now}(s)$
15.     $\mathsf{now}(s) \leftarrow \mathsf{now}(s) \setminus \{\eta\}$
16.     if ($\eta = \mathsf{false}$ or $\neg\eta \in \mathsf{old}(s)$)  return $Node\_Set$
17.     $\mathsf{old}(s_1), \mathsf{old}(s_2) \leftarrow \mathsf{old}(s) \cup \{\eta\}$
18.     $\mathsf{now}(s_1) \leftarrow \mathsf{now}_1(\eta) \cup \mathsf{now}(s)$
19.     $\mathsf{now}(s_2) \leftarrow \mathsf{now}_2(\eta) \cup \mathsf{now}(s)$
20.     $\mathsf{next}(s_1) \leftarrow \mathsf{next}_1(\eta) \cup \mathsf{next}(s)$
21.     $\mathsf{next}(s_2) \leftarrow \mathsf{next}(s)$
22.     return EPXAND($s_2$, EXPAND($s_1, Node\_Set$))
23. end function
```

On line 19, in the case that $\mathsf{now}_2(\eta)$ is $-$ then skip the line (and return EXPAND($s_1, Node\_Set$)).