Formal Verification Methods

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

Contents

1 Transition Systems 1
2 Specification Formalisms 4

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 $\neg(p_1 \vee \dots \vee p_n) \to (v := v)$ 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($\mathcal{S}, T, \Theta$)
1.     choose some initial state $s$ such that $s \vDash^{\mathcal{S}} \Theta$
2.     while ($s$ has an enabled transition)
3.         choose a transition $t$ enabled by $s$
4.         $s \leftarrow t(s)$
5.     end while
       ▷ Extend the sequence infinitely if the final state has no enabled transition
6.     repeat $s$ forever
7. 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
1.     while (true)
           ▷ wait until turn is zero
2.         wait(turn = 0)
3.         turn ← 1
       end while
   end routine
```

```
   routine PROGRAM2
1.     while (true)
           ▷ wait until turn is one
2.         wait(turn = 1)
3.         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:

$$\begin{tikzcd}
 & \mathbf{(0,1,1)} \arrow[dl] \arrow[d] & \mathbf{(1,1,1)} \arrow[d] \arrow[dr] & \\
(0,1,2) \arrow[d] & (0,2,1) \arrow[dl] \arrow[d] & (1,1,2) \arrow[d] \arrow[dr] & (1,2,1) \arrow[d] \\
(0,2,2) \arrow[dr] & (0,3,1) \arrow[d] \arrow[uur] & (1,1,3) \arrow[d] \arrow[uul] & (1,2,2) \arrow[dl] \\
 & (0,3,2) \arrow[uur] & (1,2,3) \arrow[uul] &
\end{tikzcd}$$

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 $(0,1,1) \to (0,1,2)$ and then $(0,1,2)$ is extended forever via t_4'.

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

$$t_1^i\colon v_i = 1 \longrightarrow v_i := 2, \quad t_2^i\colon v_i = 2 \longrightarrow v_i := 3, \quad t_3^i\colon v_i = 3 \longrightarrow v_i := 1$$

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.

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 $\xi = x_0 x_1 x_2 \dots$. Let us write ξ^k for the suffix $\xi^k := x_k x_{k+1} \dots$, 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 $\xi^i \vDash \psi$,

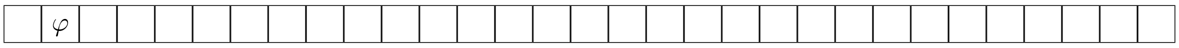
(6) $\xi^k \vDash \Box\varphi$ if $\xi^i \vDash \psi$ for every $i \geq k$,

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

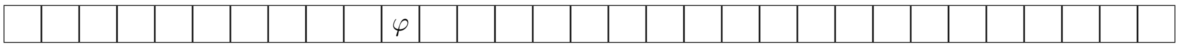
(8) $\xi^k \vDash \varphi \mathsf{V} \psi$ if for every $i \geq k$, $\xi^i \vDash \psi$; or for some $i \geq k$, $\xi^i \vDash \varphi$ and for every $k \leq j \leq i$, $\xi^j \vDash \psi$.

Intuitively we can explain the new operators as follows:

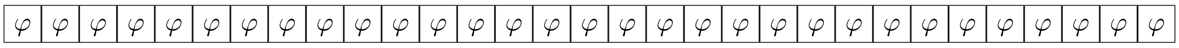
(4) $\bigcirc$ is the *nexttime* operator: $\bigcirc\varphi$ holds in the sequence $x_k x_{k+1} \dots$ if φ holds starting from the next state x_{k+1}. Visually we can view it like so:



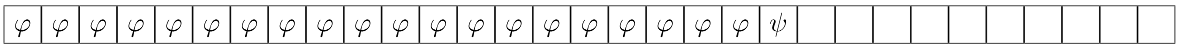
(5) $\Diamond$ is the *eventually* operator: $\Diamond\varphi$ holds in the sequence ξ provided there exists a suffix in which φ holds. Visually:



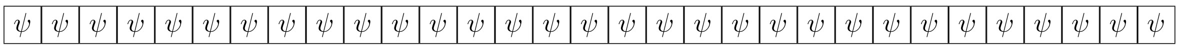
(6) $\Box$ is the *always* operator: $\Box\varphi$ holds in the sequence ξ provided it holds in every suffix of ξ. Visually:



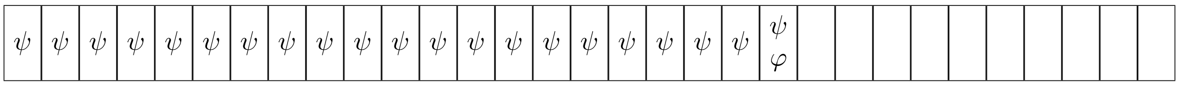
(7) U is the *until* operator: $\varphi \mathsf{U} \psi$ holds in the sequence ξ if ψ holds eventually and φ holds up until then:



(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.



Or:



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 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.