Model Checking Basics

Somesh Jha
Computer Sciences Department
University of Wisconsin
Madison, WI 53706

1 Introduction

In software engineering several formalism are in some form or another compositions of state machines. For example, *Statecharts* are simply state machines. There is value in simply writing these formal specifications down because it forces the designer to think carefully. However, in highly distributed designs subtle errors (such as deadlocks or race conditions) are very hard to catch simply by inspection. The difficulty stems from the fact that the global state space of the entire system can be very large and exhibit very complex behaviors. Therefore, there is a need for automatic analysis of specifications expressed as composition of state machines. *Model checking* is a technique for automatically analyzing whether a model of a distributed system has a desired property, e.g., absence of deadlocks.

Model checking takes as its input a *formal model* of the system and a *property* expressed in temporal logic. Temporal logics are logics that have a notion of *time*. Using sophisticated state space exploration techniques a *model checker* verifies that the model satisfies the desired property. If the property turns out to be false in the model, most model checkers output a *counter-example* or a trace of states in the model that shows "why" the property does not hold. Figure 1 gives a schematic description of a model-checker.

Next we describe the two inputs to the model checker. The discussion is kept at an abstract level. We will discuss specific details when we describe the model checker `NuSMV`.

2 Describing the model

A *model* M has two components V and R described below:

- **Variables**
  V is the set of state variables in the model. If the model has n state variables, we will denote them by $v_1, \cdots, v_n$. Each state variable v_i has an associated domain D_i. Notice that the entire state space of the system is simply the Cartesian product of the domains $D_1, \cdots, D_n$ (denoted by $\prod_{i=1}^{n} D_i$). A *state* is simply an assignment to the variables in the set V from their domain. For example, consider a model that has two variables x and y with domains $\{a, b, c\}$ and $\{0, 1\}$ respectively. Then $(x = b, y = 1)$ is a state of the model. The set of all states is called the *state space* of the model and will be denoted by S.

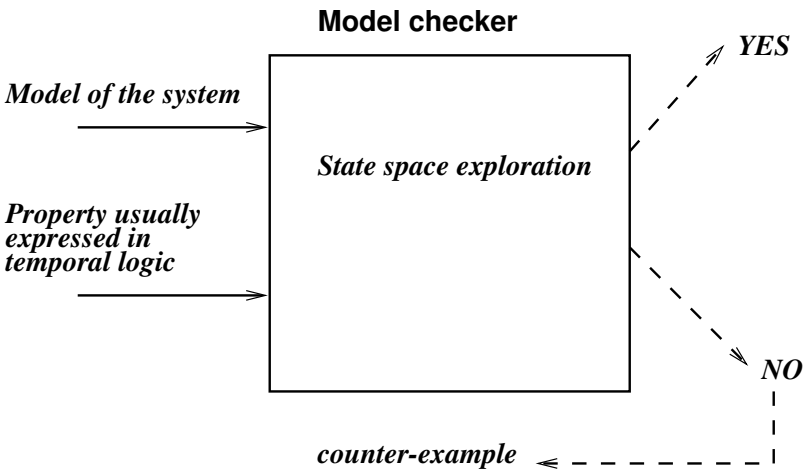

Figure 1: Schematic description of model checking

- *Transition relation*
  R is the transition relation. Formally, R is the relation from S to S (denoted as $R : S \leftrightarrow S$). If $(s, s') \in R$, this means that from state s it is *possible* or *valid* to transition to state s'. The transition relation R encapsulates all possible/valid transitions in the model.

Usually, a set of *initial states* $I \subseteq S$ is also specified. The model M can start in any of the initial states I. Therefore, we will write a model M as (V, R, I) emphasizing the fact that it has three components, i.e., the set of variables, transition relation, and a set of initial states. The specific language used to describe the model is not very important to describe the concepts. We will describe a specific modeling language later. States of a model M were defined before. A *path* is a finite or infinite sequence of states $(s_0, s_1, \cdots, s_i, \cdots)$ such that (s_i, s_{i+1}) is a valid transition in the model. In other words a path describes a sequence of states were each step corresponds to a valid transition in the model. Usually we will denote a path using the Greek symbol π. Given a path $\pi = (s_0, s_1, \cdots, s_i, \cdots)$, π^i denotes the suffix of π starting at the i-th state, or given by the following sequence:

$$(s_i, s_{i+1}, \cdots,)$$

Figure 2 shows a small model (can you guess what the model does?). A path π through the model is shown below:

$$((x = 0, y = -1), (x = 0, y = 0), (x = 0, y = 1), (x = 1, y = 1), (x = 0, y = 0))$$

Path π^2 starts at the second state (remember we start counting from zero) and is shown below:

$$((x = 0, y = 1), (x = 1, y = 1), (x = 0, y = 0))$$

3 Temporal logic

In temporal logic one can talk about "time" in addition to atomic properties. As we will see later, it is important to have the notion of time to express properties about reactive systems. We will describe a powerful temporal logic called the *Computation Tree Logic* or *CTL** for short.

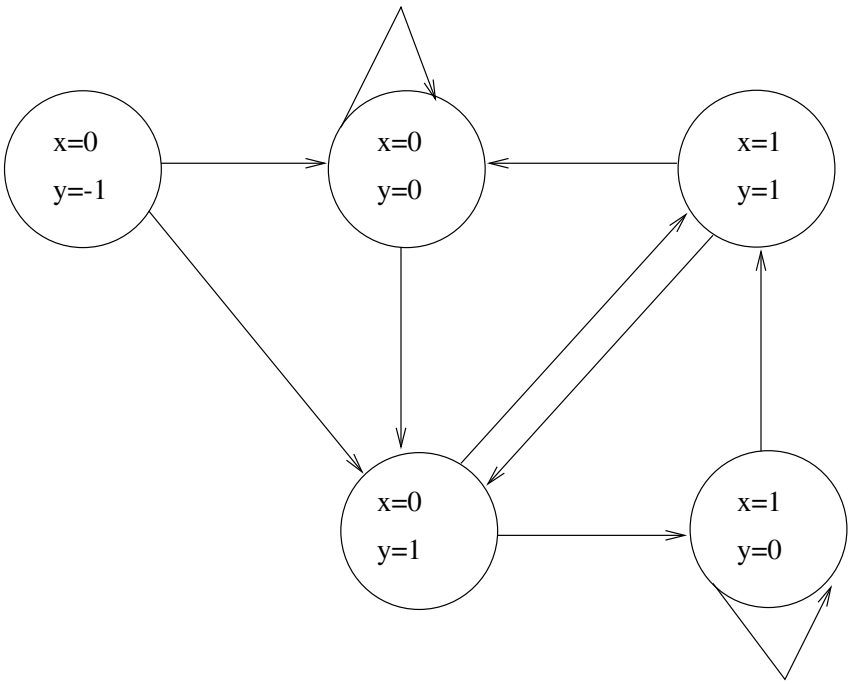

Figure 2: A small example

First, we describe *atomic properties* for describing "local" properties about states. Throughout this section consider a model $M = (V, I, R)$, where there are n state variables $V = \{v_1, \cdots, v_n\}$. Let D_i be the domain associated with variable v_i. A *basic atomic property* has the following form:

- $v_i = x$, $v_i \in S$, $v_i < x$, and $v_i > x$ where x is another variable from the set V or a constant from the domain D_i associated with the variable v_i and S is a subset of the domain D_i associated with x.

A basic atomic property is an *atomic property*. More complicated atomic properties can be constructed from other atomic properties using conjunction (denoted by $\wedge$), disjunction (denoted by $\vee$), and negation (denoted by $\neg$). An example of an atomic property is given below:

$$p \;=\; (x = y) \wedge \neg(z > 3) \tag{1}$$

Recall that a state s is simply an assignment to all the variables $v_1, \cdots, v_n$. Suppose there is a program with three variables x, y, and z. Consider the following two states:

$$\begin{aligned} s_1 &= (x = 3, y = 3, z = 2) \\ s_2 &= (x = 3, y = 2, z = 2) \end{aligned}$$

Consider the atomic property p shown in equation 1. The atomic property p shown in equation 1 is true in state s_1 and false in state s_2 (why?). We denote this fact by $s_1 \models p$ (or s_1 satisfies p) and $s_2 \not\models p$ (or s_2 does not satisfy p). Hence given an atomic property p and a state s one can decide whether or not s satisfies p.

CTL* has two types of formulas *state formulas* (to express properties about states) and *path formulas* (to express properties about paths). Let AP be the set of atomic properties. Grammars

$$
\begin{array}{lcc}
SF & \rightarrow & AP \\
SF & \rightarrow & \{\neg SF \mid SF \wedge SF \mid SF \vee SF\} \\
SF & \rightarrow & \{\mathbf{A}(PF) \mid \mathbf{E}(PF)\}
\end{array}
$$

Figure 3: Grammar for generating state formulas

$$
\begin{array}{lcc}
PF & \rightarrow & SF \\
PF & \rightarrow & \{\neg PF \mid PF \wedge PF \mid PF \vee PF\} \\
PF & \rightarrow & \{\mathbf{X}\ PF \mid \mathbf{F}\ PF \mid \mathbf{G}\ PF \mid PF\ \mathbf{U}\ PF \mid PF\ \mathbf{R}\ PF\}
\end{array}
$$

Figure 4: Grammar for generating path formulas

for generating state and path formulas in *CTL** are shown in Figure 3 and 4 respectively. SF and PF are non-terminals that derive state and path formulas respectively. Recall that AP is the set of atomic formulas.

Exercise 1 Consider the formula f given below:

$$f \quad = \quad \mathbf{A}((x = 2)\ \mathbf{U}\ (\mathbf{E}(y > 3)))$$

Is f a path or state formula? Use the grammars shown in Figures 3 and 4 to derive f.

We have given the syntax of *CTL**. Next we turn to the semantic of *CTL**, i.e., given a model $M = (V, R, I)$ and a *CTL** formula f, how does one one determine whether M satisfies f (denoted by $M \models f$)? First we define the "satisfaction" relation for states and paths and then for models. In general, for the statements given below $M, s \models f$ means that a state s in the model M satisfies the state formula f and $M, \pi \models f$ means that the path π satisfies a path formula f. Remember that a state formula is interpreted over states and a path formula is interpreted over paths.

$$
\begin{array}{rcl}
M, s \models p & \Leftrightarrow & s \models p \\
M, s \models \neg f_1 & \Leftrightarrow & M, s \not\models f_1 \\
M, s \models f_1 \wedge f_2 & \Leftrightarrow & M, s \models f_1 \text{ and } M, s \models f_2 \\
M, s \models f_1 \vee f_2 & \Leftrightarrow & M, s \models f_1 \text{ or } M, s \models f_2 \\
M, s \models \mathbf{E}(f_1) & \Leftrightarrow & \text{there is a path } \pi \text{ from } s \text{ such that } M, \pi \models g_1 \\
M, s \models \mathbf{A}(f_1) & \Leftrightarrow & \text{for every path } \pi \text{ from } s\ M, \pi \models g_1 \text{ holds} \\
M, \pi \models f_1 & \Leftrightarrow & s \text{ is the first state of } \pi \text{ and } M, s \models f_1 \\
M, \pi \models \neg g_1 & \Leftrightarrow & M, \pi \not\models g_1 \\
M, \pi \models g_1 \wedge g_2 & \Leftrightarrow & M, \pi \models g_1 \text{ and } M, \pi \models g_2 \\
M, \pi \models g_1 \vee g_2 & \Leftrightarrow & M, \pi \models g_1 \text{ or } M, \pi \models g_2 \\
M, \pi \models \mathbf{X}\ g_1 & \Leftrightarrow & M, \pi^1 \models g_1 \\
M, \pi \models \mathbf{F}\ g_1 & \Leftrightarrow & \text{there exists a } k \geq 0 \text{ such that } M, \pi^k \models g_1
\end{array}
$$

$$
\begin{array}{lcl}
M, \pi \models g_1 \ \mathbf{U} \ g_2 & \Leftrightarrow & \text{there exists a } k \geq 0 \text{ such that} \\
 & & M, \pi^k \models g_2 \text{ and for all } 0 \leq j < k, M, \pi^j \models g_1 \\
M, \pi \models g_1 \ \mathbf{R} \ g_2 & \Leftrightarrow & \text{for all } j \geq 0, \text{ if for every} \\
 & & i < j, M, \pi^i \not\models g_1 \text{ then } M, \pi^j \models g_2
\end{array}
$$

Notice that the semantics given above is with respect to a specific state and a path in the model. A model $M = (V, R, I)$ is *said to satisfy* a state formula f if for all initial states $s \in I$, $M, s \models f$, or in other words all initial states of the model satisfy the formula f. Next we try to explain the semantics of CTL in an intuitive manner. Here is an explanation of the various operators that can appear in a path formula.

- **X** ("next time") requires that a property hold in the second state of the path.
- **F** ("eventually" or "in the future") operators is used to assert that a property will hold at some state on the path.
- **G** ("always" or "globally") specifies that a property holds at every state on the path.
- **U** ("until") operator is a bit more complicated since it is used to combine two properties. It holds if there is a state on the path where the second property holds, and at every preceding state on the path, the first property holds.
- **R** ("releases") also combines two properties, and is the logical dual of the **U** operator. It requires that the second property holds along the path up to an including the first state where the first property holds. In other words, once the first property becomes true, the second property is "released" of its commitment, i.e., does not have to be true anymore.

Make sure that you understand the explanations given above. Try to relate the explanations given above to formal semantics provided earlier. Now we will give a procedure to translate a *CTL** formula into a natural language form. This will aid you in understanding the formulas. Let $TR(f)$ denote the translation of formula f into english (well sort of english). Here is the recursive definition of f:

- $TR(p)$
  (state satisfies $TR(p)$)
- $\neg f_1$
  (state does not satisfy $TR(f_1)$)
- $f_1 \wedge f_2$
  (state satisfies $TR(f_1)$ and $TR(f_2)$)
- $f_1 \vee f_2$
  (state satisfies $TR(f_1)$ or $TR(f_2)$)

- $\mathbf{A}(f)$
  (all paths starting from the state satisfy $TR(f)$)
- $\mathbf{E}(f)$
  (there exists a path starting from the state satisfying $TR(f)$)
- f_1
  (path satisfies $TR(f_1)$)
- $\neg g_1$
  (path does not satisfy $TR(g_1)$)
- $g_1 \vee g_2$
  (path satisfies $TR(g_1)$ or $TR(g_2)$)
- $g_1 \wedge g_2$
  (path satisfies $TR(g_1)$ and $TR(g_2)$)
- $\mathbf{X}\ g_1$
  (in the next time on a path $TR(g_1)$ holds)
- $\mathbf{F}\ g_1$
  (eventually on the path $TR(g_1)$ holds)
- $\mathbf{G}\ g_1$
  (globally on the path $TR(g_1)$ holds)
- $g_1\ \mathbf{U}\ g_2$
  ($TR(g_2)$ eventually holds on the path and until that time $TR(g_1)$ holds)
- $g_1\ \mathbf{R}\ g_2$
  ($TR(g_2)$ holds along the path up to and including the first state where $TR(g_1)$ holds)

Consider the following formula:

$$\mathbf{AG}(Req \rightarrow \mathbf{AF}\ Ack)$$

Translating the formula using the recipe given above we get following natural language phrase:

```
(all paths starting from the state satisfy
   (globally on the path
      (if ``Req'' is true then
         (all paths starting from the state satisfy
             (eventually on the path ``Ack'' holds)
         )
       )
    holds )
)
```

The english phrase given above can be paraphrased as (check this!)

Always if a request occurs, then it is always eventually acknowledged.

$$PF \quad \rightarrow \quad \{\mathbf{X}\ SF \mid \mathbf{F}\ SF \mid \mathbf{G}\ SF \mid SF\ \mathbf{U}\ SF \mid SF\ \mathbf{R}\ SF\}$$

Figure 5: Grammar for generating path formulas for *CTL*

$$\begin{array}{lll} PF & \rightarrow & AP \\ PF & \rightarrow & \{\neg PF \mid PF \wedge PF \mid PF \vee PF\} \\ PF & \rightarrow & \{\mathbf{X}\ PF \mid \mathbf{F}\ PF \mid \mathbf{G}\ PF \mid PF\ \mathbf{U}\ PF \mid PF\ \mathbf{R}\ PF\} \end{array}$$

Figure 6: Grammar for generating path formulas for *LTL*

3.1 *CTL* and *LTL*

CTL and *LTL* are important fragments of the powerful logic *CTL**. Both *CTL* and *LTL* are "less powerful" than *CTL**. *CTL* is a restricted subset of *CTL** where each of the temporal operators **X**,**F**,**G**,**U**, and **R** have to be immediately preceded by a path quantifier **A** or **E**. The grammar describing the state formula for the logic *CTL* is the same as given in Figure 3. The grammar for describing path formulas in the case of *CTL* is shown in Figure 5.

Linear temporal logic (or *LTL*) for short) where the state formulas are of the form $\mathbf{A}f$, where f is a path formula. In other words, the only production describing the state formula in the case of *LTL* is:

$$SF \quad \rightarrow \quad \mathbf{A}\ PF$$

The grammar describing path formula for *LTL* is shown in Figure 6.

Check that $\mathbf{A}(\mathbf{FG}p)$ is not a *CTL* formula but a *LTL* formula. It can be shown that there does not exist a *CTL* formula that is "equivalent" to the *LTL* formula $\mathbf{A}(\mathbf{FG}p)$. Similarly, there is not *LTL* formula that is equivalent to the *CTL* formula $\mathbf{AG}(\mathbf{EF}p)$. The disjunction of these two formulas

$$\mathbf{A}(\mathbf{FG}p) \vee \mathbf{AG}(\mathbf{EF}p)$$

is a *CTL** formula that is not expressible in either *CTL* or *LTL*. In other words, *CTL* and *LTL* are strict subsets of *CTL**, and *CTL* and *LTL* are incomparable.

3.2 Typical specifications

Some typical specifications that might arise during the verification process are described next. Make sure that the explanations provided with each specification are compatible with the semantics given earlier.

- $\mathbf{EF}(Started \wedge \neg Ready)$: It is possible to get to a state where $Started$ holds but $Ready$ does not hold.
- $\mathbf{AG}(Req \rightarrow \mathbf{AF}Ack)$: If a request occurs, then it will be eventually acknowledged.
- $\mathbf{AG}(\mathbf{AF}DeviceEnable)$: The proposition $DeviceEnabled$ holds infinitely often on every computation path.
- $\mathbf{AG}(\mathbf{EF}Restart)$: From any state it is possible to get to the $Restart$ state.

3.3 Relationships between formulas

Most of the formulas we will encounter will be written in *CTL*. There are ten basic operators in *CTL*.

- **AX** and **EX**.
- **AF** and **EF**.
- **AG** and **EG**.
- **AU** and **EU**.
- **AR** and **ER**.

Each of the ten operators can be expressed in terms of the three operators **EX**, **EG**, **EU**. So there are only three basic operators and the rest is just "syntactic sugar".

- **AX** $f \;=\; \neg$**EX**$(\neg f)$
- **EF** $f \;=\;$ **E**$[true$ **U** $f]$
- **AG** $f \;=\; \neg$**EF**$(\neg f)$
- **AF** $f \;=\; \neg$**EG**$(\neg f)$
- **A**$[f$ **U** $g] \;=\; \neg$**E**$[\neg g$ **U** $\neg f \wedge \neg g] \wedge \neg$**EG**$\neg g$
- **A**$[f$**R**$g] \;=\; \neg$**E**$[\neg f$ **U** $\neg g]$
- **E**$[f$**R**$g] \;=\; \neg$**A**$[\neg f$ **U** $\neg g]$

Let us reason about the third equation. We will refer to the equations given above as the *duality equations*. Suppose **AG** f is true in a state s of the model. This means that on every path π starting from the state s the formula f holds globally. In other words, $\neg f$ is never true on any path π starting from the state s, or **EF**$\neg f$ is not true in state s. You should reason about the other equations in a similar manner and convince yourself that they are true.