References on Model Checking: *Logic in Computer Science: Modelling and Reasoning about Systems, 2nd Edition* by Huth and Ryan; *Model Checking* by Clarke, Grumberg and Peled.

15.1 Introduction to Model Checking

In the past: Goal was to prove that your system or program was completely correct. This turned out to be pie in the sky because of the following problems:

1. Facts about programs, including the simple question of whether or not a program eventually halts, are usually **undecidable**.
2. There are infinitely many possible inputs and runs to validate.
3. In order to prove that the program is correct, we need to prove that it exactly matches its specification. Thus, we need a perfect specification. Coming up with a perfect formal specification is **essentially the same task** as writing a program that implements it.

Model Checking Idea: As we are in the process of designing our system or program, we build and modify our design in a formal design language or programming languages. During the design process we can specify some desirable properties. For example:

1. (in the design of a subway system) The doors do not open between stations.
2. We never divide by 0.
3. We never follow a null pointer.

The model checker automatically will tell us, "yes", your condition is always true; or "no" and "here is a counterexample execution sequence of your design". The designer can then check whether the counter example is an error in the design, or whether the specified property was the problem, e.g., maybe in certain emergencies, when someone has pushed the emergency exit button, the doors correctly open between stations.

Pentium FDIV bug: In 1994, a bug was discovered in the floating point divide function of the Intel Pentium P5 processor. This led to an expensive recall and was also a factor in Intel and most other hardware complanies now mandating model checking in their design process.

15.2 Formal Setting of Model Checking

Model Checking can refer to finite-state systems or infinite-state systems. Hardware is typically finite state and software can be either. We will concentrate on finite-state systems.

We now describe the set of possible models called **transition systems** and the **specification languages**. We will use **temporal logic** and specifically today, Linear Temporal Logic (LTL).

Any model checking system will automatically construct a transition system from the given design. (The mapping from the design to the transition system is similar to compilation.)

A key complexity issue that must be coped with is called the **State Explosion Problem**, namely that for a design of size n, the number of states in the transition system tends to be exponentially larger, i.e., size 2^n or greater.

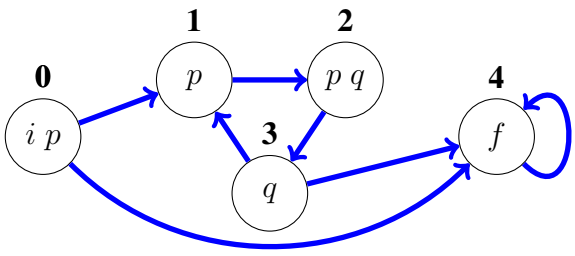
Definition 15.1 [Transition System] A Transition System, $\mathcal{T} = (S, R, L)$, consists of a set of states, S, the transition relation, $R \subseteq S^2$, a set of atomic propositions, AP and a labeling function L mapping each state, $s \in S$, to $L(s)$, the set of atomic propositions that it satisfies. Thus, $L : S \to \wp(\text{AP})$ □

Intuitively, a state $s \in S$, is a global state describing everything about the system, e.g., for hardware, the setting of every gate or latch, and for software, the contents of every memory location and register. A transition is an atomic transition, i.e., execution of of a single step or instruction.

Proviso: we shall assume that every state of every transition system always has at least one outgoing transition. This is because temporal logic refers to runs consisting of infinite paths through the transition system.

Example 15.2 Consider the following transition system,

$\mathcal{T}_0 = (S_0, R_0, L_0)$



□

Temporal Logic

There are two kinds of properties that we typically want to express in Model Checking:

Safety Property: It is impossible to reach a bad state, e.g., we never try to open the doors between stations; we never divide by 0, we never try to access a null pointer.

Liveness: It is always the case that something good will eventually happen, e.g., every appropriate request for data, printing, opening a connection, etc. will eventually be granted.

Temporal Logic was popularized by Amir Pneuli. The advantages of this language over first-order logic is its simplicity: properties are **easy to specify** and **easy to check**.

Definition 15.3 [Syntax of LTL] Given a set of atomic propositions, inductively define the LTL formulas, form(LTL), as follows:

base case: $p \in \text{form(LTL)}$ for $p \in \text{AP}$, an atomic proposition

inductive cases: for $\alpha, \beta \in \text{form(LTL)}$, the following are in form(LTL):

1. $\neg\alpha$
2. $(\alpha \vee \beta)$
3. $\mathbf{X}\alpha$ "α is true at the **next** step."
4. $\mathbf{G}\alpha$ "α is true **globally**, i.e., at all times now and in the future."
5. $\mathbf{F}\alpha$ "α is true in the **future**, i.e., now or at some time in the future."
6. $(\alpha \mathbf{U} \beta)$ "α is true **until** β first becomes true, which will happen."

□

Definition 15.4 [paths] The semantics of LTL is defined with respect to paths of transition systems. Given a transition system, $\mathcal{T} = (S, R, L)$, a **path**, $\pi \in \text{path}(\mathcal{T})$ is an infinite sequence of states, $\pi = s_0, s_1, s_2, \ldots$, such that for all i, $(s_i, s_{i+1}) \in R$.

Given a path $\pi = s_0, s_1, s_2, \ldots$, let $\pi[i] \stackrel{\text{def}}{=} s_i$, i.e., the ith state in the path.

Let $\pi^i \stackrel{\text{def}}{=} s_i, s_{i+1}, s_{i+2}, \ldots$ be the path π with the first i states removed. □

Definition 15.5 [Semantics of LTL] For any $\alpha \in \text{form(LTL)}$, any transition system, $\mathcal{T}$, and any $\pi \in \text{path}(\mathcal{T})$, we inductively define what it means for $(\mathcal{T}, \pi)$ to satisfy α.

0. $(\mathcal{T}, \pi) \models p$ iff $p \in L(\pi[0])$
1. $(\mathcal{T}, \pi) \models \neg\alpha$ iff $(\mathcal{T}, \pi) \not\models \alpha$
2. $(\mathcal{T}, \pi) \models (\alpha \vee \beta)$ iff $(\mathcal{T}, \pi) \models \alpha$ or $(\mathcal{T}, \pi) \models \beta$
3. $(\mathcal{T}, \pi) \models \mathbf{X}\alpha$ iff $\pi^1 \models \alpha$
4. $(\mathcal{T}, \pi) \models \mathbf{G}\alpha$ iff $\forall i \geq 0\ (\mathcal{T}, \pi^i) \models \alpha$
5. $(\mathcal{T}, \pi) \models \mathbf{F}\alpha$ iff $\exists i \geq 0\ (\mathcal{T}, \pi^i) \models \alpha$
6. $(\mathcal{T}, \pi) \models (\alpha \mathbf{U} \beta)$ iff $\exists i \geq 0\ ((\mathcal{T}, \pi^i) \models \beta \wedge \forall j < i\ (\mathcal{T}, \pi^j) \models \alpha)$

When $\mathcal{T}$ is understood, we may write, "$\pi \models \alpha$" as an abbreviation of $(\mathcal{T}, \pi) \models \alpha$. □

For example, consider the following paths from the transition system $\mathcal{T}_0$ of Example **??**:

$$\begin{aligned}
\pi &= s_0, s_1, s_2, s_3, s_4, \overline{s_4,} \quad \text{i.e., infinitely repeat } s_4 \\
\sigma &= s_0, s_1, s_2, s_3, s_1, s_2, s_3, \overline{s_4,} \\
\rho &= s_0, \overline{s_1, s_2, s_3,} \\
\gamma &= s_0, \overline{s_4,} \ldots
\end{aligned}$$

Exercise 15.6 Use Def. **??** to determine which of the above paths satisfy the following formulas.

$$\begin{aligned}
\alpha &\equiv p \\
\beta &\equiv \mathbf{F}p \\
\delta &\equiv (p\mathbf{U}q) \\
\psi &\equiv ((p \wedge \neg q)\mathbf{U}f) \\
\varphi &\equiv \mathbf{GF}q
\end{aligned}$$

□