Model Checking and Symbolic Execution

Ahmed Rezine

IDA, Linköpings Universitet

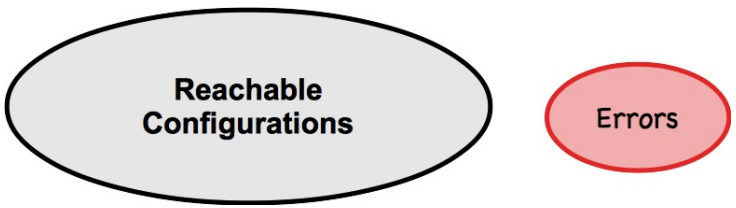
Hösttermin 2021

Outline

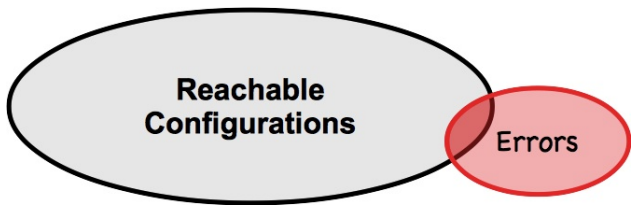
Outline

Program verification and Approximations

We often want to answer whether a program is **safe** or not (i.e., has some erroneous reachable configurations or not):



Safe Program



Unsafe Program

Program Verification and Approximations

- Answering whether all runs are error-free is very hard
- Suppose we could build a program which takes as input arbitrary computer programs and one of their lines. If such a program was able to always terminate answering correctly whether the line is reachable, then we would always be able to answer whether a Turing machine halts.
- This problem is proven to be undecidable, i.e., there is no algorithm that is guaranteed to terminate and to give an exact answer to the problem.

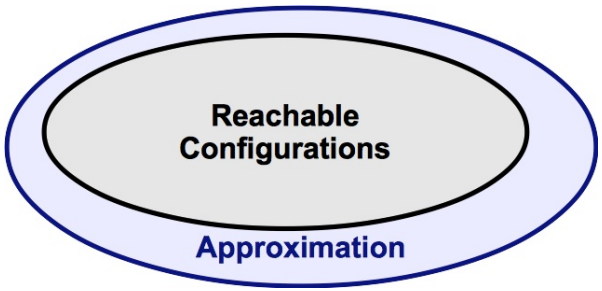
Program Verification and Approximations

Instead, work with approximations:

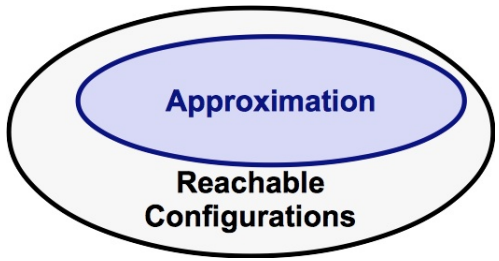
- An algorithm is **sound** in the case where each time it reports the program is safe wrt. some errors, then the original program is indeed safe wrt. those errors
- An algorithm is **complete** in the case where each time it is given a program that is safe wrt. some errors, then it does report it to be safe wrt. those errors

Program Verification and Approximations

- The idea is then to come up with efficient approximations and algorithms to give correct answers in as many cases as possible.



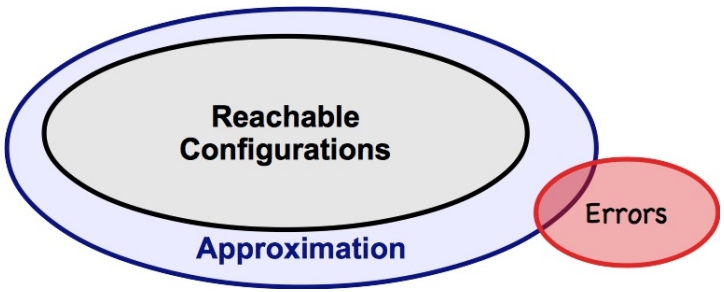
Over-approximation



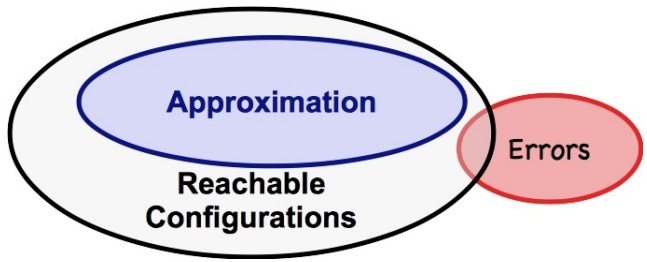
Under-approximation

Program Verification and Approximations

- A sound analysis cannot give **false negatives**
- A complete analysis cannot give **false positives**



False Positive



False Negative

In this lecture

We will briefly introduce two approaches to algorithmic verificaiton:

- Model checking: exhaustive, aims for soundness
- Symbolic execution: partial, aims for completeness

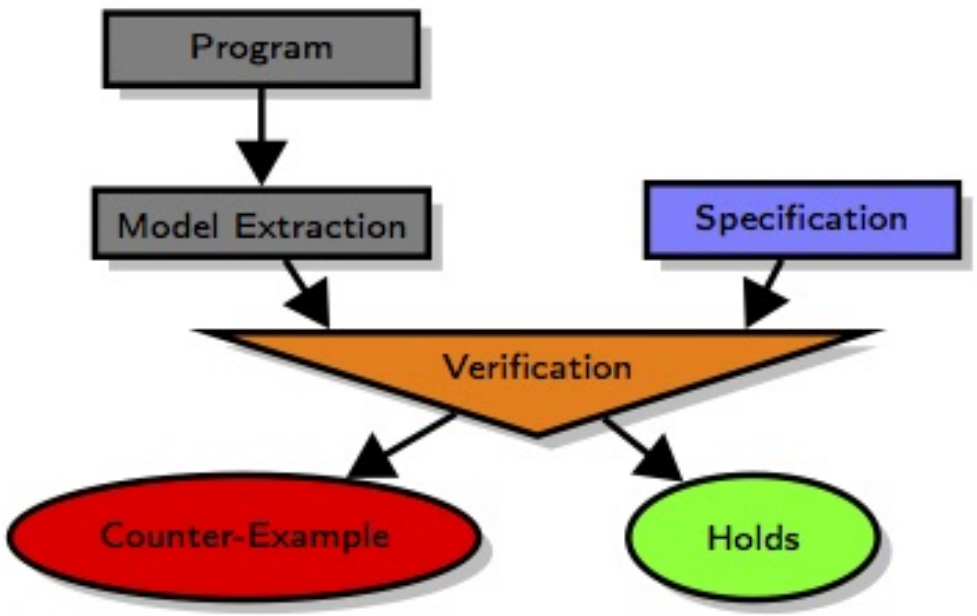
Outline

Model checking

- Model checking is a push button verification approach
- Given:
  - a model M of the system to be verified, and
  - a correctness property Φ to be checked: absence of deadlocks, livelocks, starvation, violations of constraints/assertions, etc
- The model checking tool returns:
  - a counter example in case M does not model Φ, or
  - a mathematical guaranty that M does model Φ

Model Checking: Verification vs debugging

- Model checking tools are used both:
  - To establish correctness of a model M with respect to a correctness property Φ
  - More importantly, to find bugs and errors in M early during the design



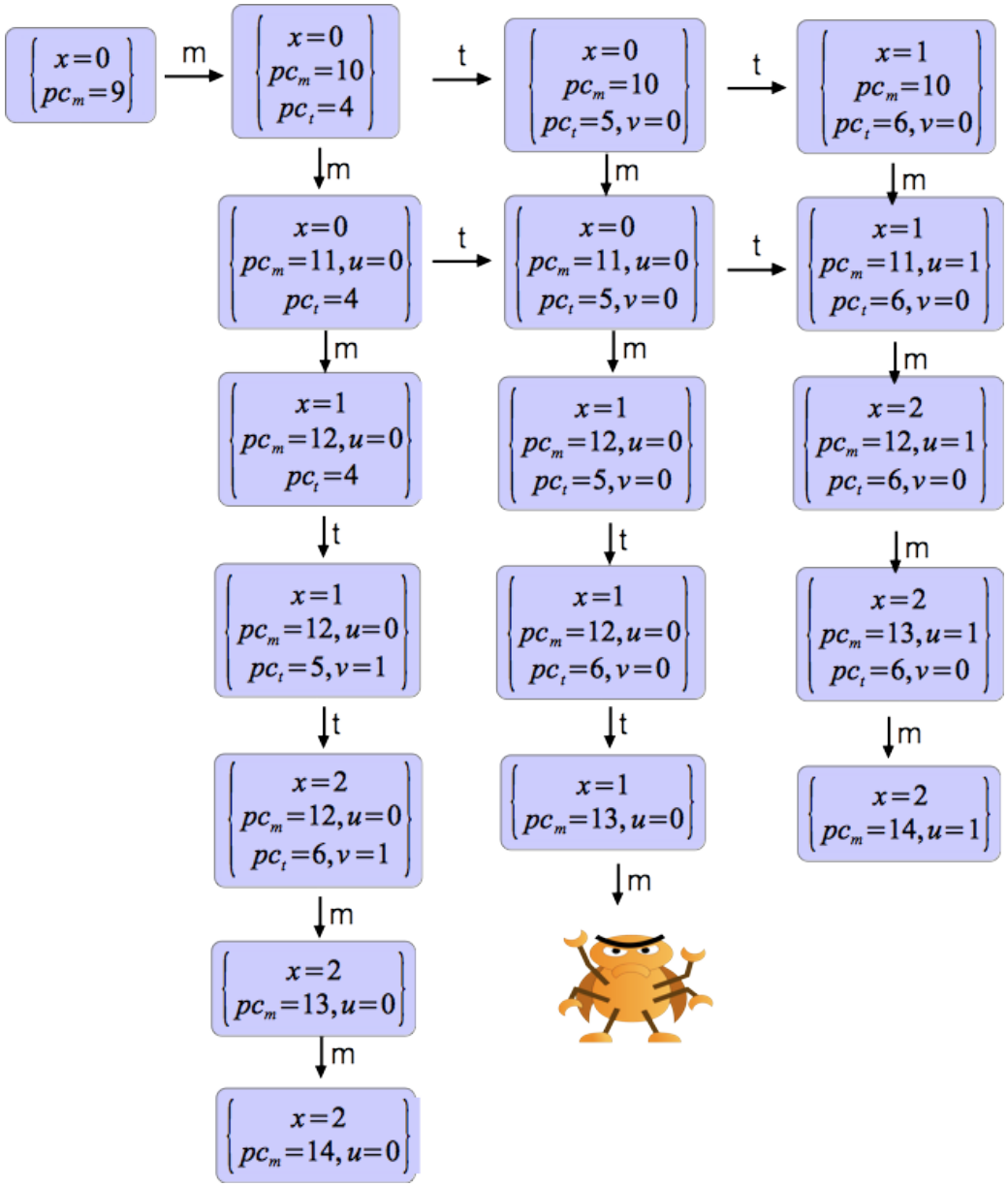
M as a Kripke structure

Assume a set of atomic propositions AP. A *Kripke structure* M is a tuple (S, S_0, R, L) where:

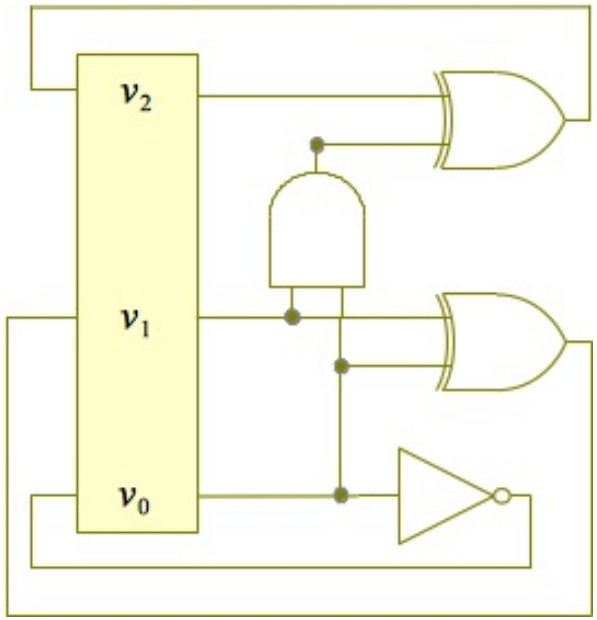
1. S is a finite set of states
2. $S_0 \subseteq S$ is the set of initial states
3. $R \subseteq S \times S$ is the transition relation s.t. for any $s \in S$, $R(s, s')$ holds for some $s' \in S$
4. $L : S \to 2^{AP}$ labels each state with the atomic propositions that hold on it.

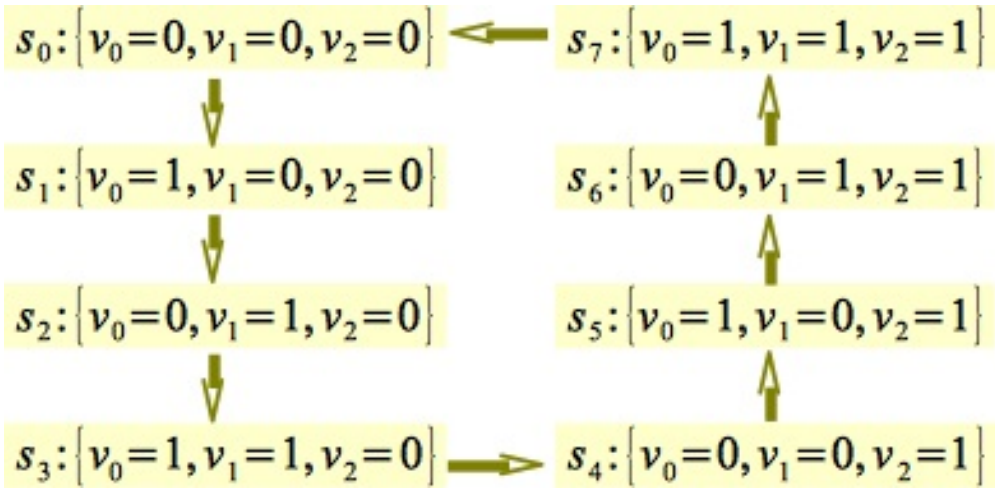
Programs as Kripke structures

```
1  int x = 0;
2
3  void thread(){
4    int v = x;
5    x = v + 1;
6  }
7
8  void main(){
9    fork(thread);
10   int u = x;
11   x = u + 1;
12   join(thread);
13   assert(x == 2);
14 }
```



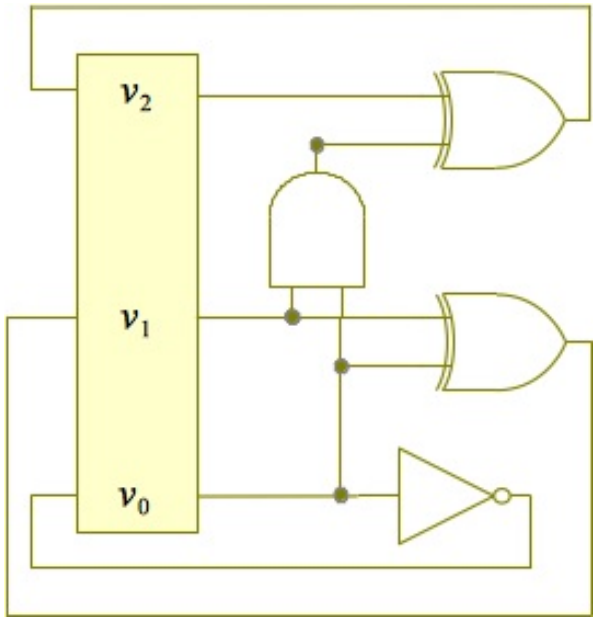
Synchronous circuits as Kripke structures

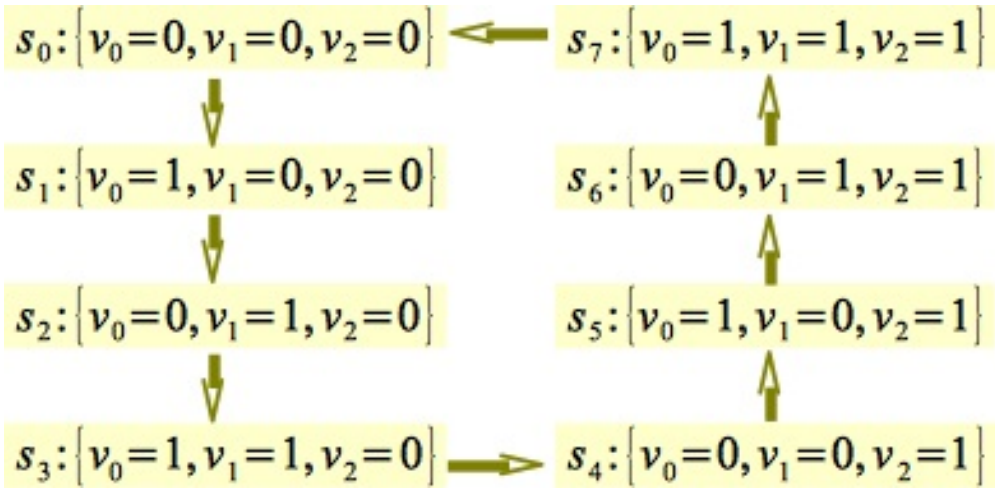




$$v_0' = \neg v_0 \quad (1)$$
$$v_1' = v_0 \oplus v_1 \quad (2)$$
$$v_2' = (v_0 \wedge v_1) \oplus v_2 \quad (3)$$

Synchronous circuits as Kripke structures





$$v_0' = \neg v_0 \quad (1)$$
$$v_1' = v_0 \oplus v_1 \quad (2)$$
$$v_2' = (v_0 \wedge v_1) \oplus v_2 \quad (3)$$

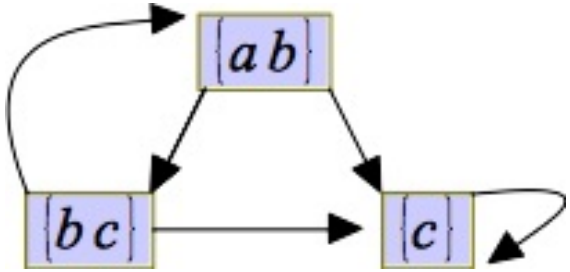
Asynchronous circuits handled using a disjunctive R instead of a conjunctive one like for synchronous circuits.

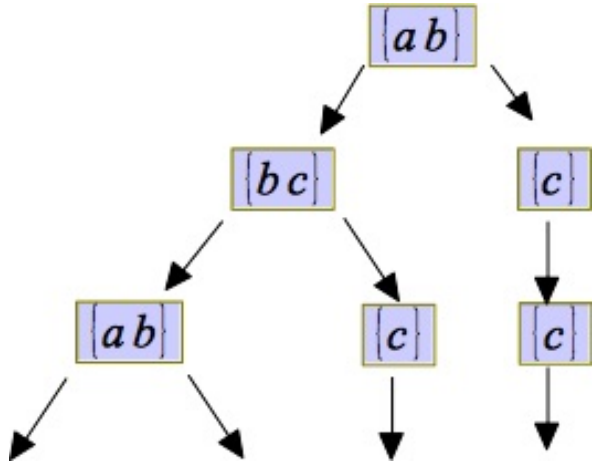
Temporal Logics

- Temporal logics are formalisms to describe sequences of transitions
- Time is not mentioned explicitly (in today's lecture)
- Instead, temporal operators are used to express that certain states are:
  - never reached
  - eventually reached
  - more complex combinations of those

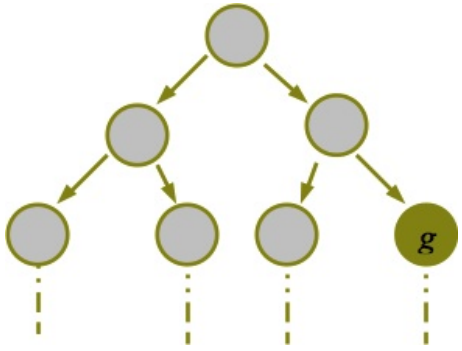
Computation Tree Logic (CTL)

Computation trees are obtained by unwinding the Kripke structure

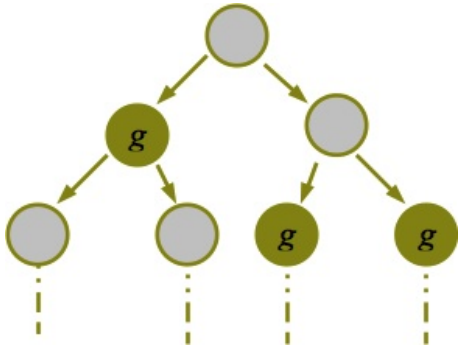




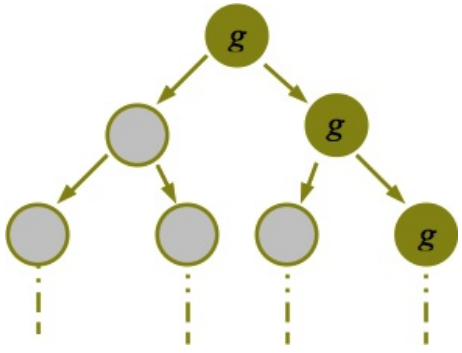
Computation Tree Logic (CTL)



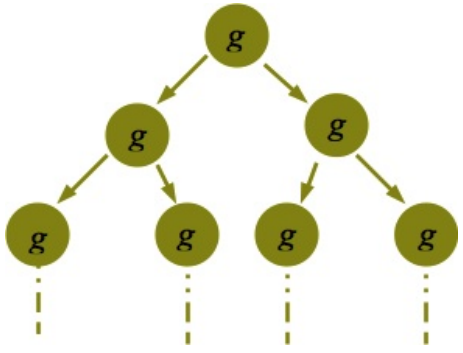
$M, s_0 \models \mathbf{EF}\ g$



$M, s_0 \models \mathbf{AF}\ g$

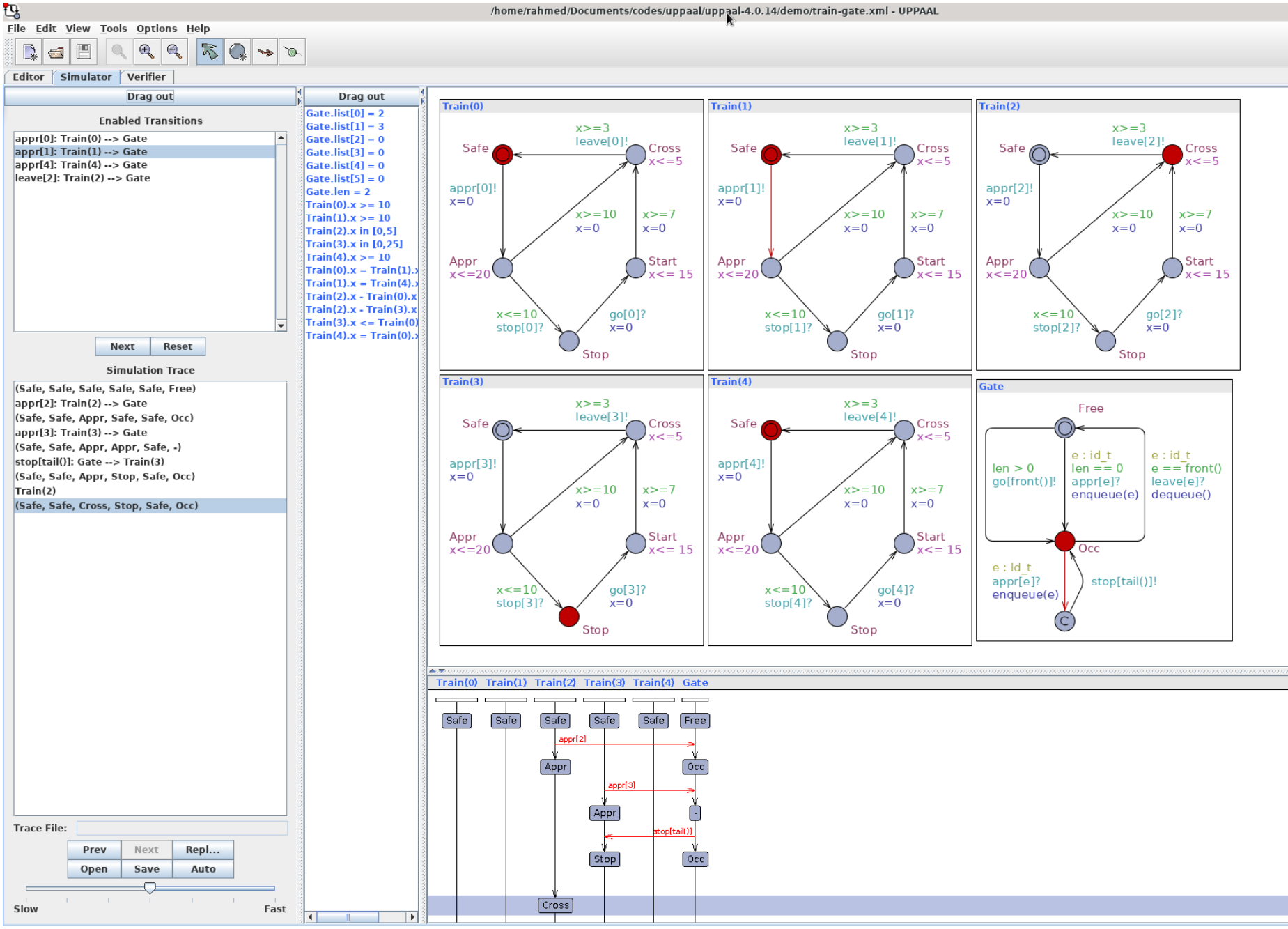


$M, s_0 \models \mathbf{EG}\ g$



$M, s_0 \models \mathbf{AG}\ g$

The UPPAAL model checker



Outline

Testing

- Most common form of software validation
- Explores only one possible execution at a time
- For each new value, run a new test.
- On a 32 bit machine, `if(i==2021) bug()` would require 2^{32} different values to make sure there is no bug.
- The idea in symbolic testing is to associate **symbolic values** to the variables

Symbolic Testing

- Main idea by JC. King in "Symbolic Execution and Program Testing" in the 70s
- Use symbolic values instead of concrete ones
- Along the path, maintain a *Path Constraint* (PC) and a symbolic state (Σ)
- PC collects constraints on variables' values along a path,
- Σ associates variables to symbolic expressions,
- We get concrete values if PC is satisfiable
- The program can be run on these values
- Negate a condition in the path constraint to get another path

Symbolic Execution: a simple example

- Can we get to the ERROR? explore using SSA forms.
- Useful to check array out of bounds, assertion violations, etc.

```
1  foo(int x,y,z){
2    x = y - z;
3    if(x==z){
4      z = z - 3;
5      if(4*z < x + y){
6        if(25 > x + y) {
7          ...
8        }
9        else{
10         ERROR;
11       }
12     }
13   }
14   ...
```

$PC_1 = true$ $\quad x \mapsto x_0, y \mapsto y_0, z \mapsto z_0$

$PC_2 = PC_1$ $\quad x \mapsto x_0, y \mapsto y_0, z \mapsto z_0$

$PC_3 = PC_2 \wedge x_1 = y_0 - z_0$ $\quad x \mapsto y_0 - z_0, y \mapsto y_0, z \mapsto z_0$

$PC_4 = PC_3 \wedge x_1 = z_0$ $\quad x \mapsto y_0 - z_0, y \mapsto y_0, z \mapsto z_0$

$PC_5 = PC_4 \wedge z_1 = z_0 - 3$ $\quad x \mapsto y_0 - z_0, y \mapsto y_0, z \mapsto z_0 - 3$

$PC_6 = PC_5 \wedge 4 * z_1 < x_1 + y_0$ $\quad x \mapsto y_0 - z_0, y \mapsto y_0, z \mapsto z_0 - 3$

$PC_{10} = PC_6 \wedge 25 \leq x_1 + y_0$ $\quad x \mapsto y_0 - z_0, y \mapsto y_0, z \mapsto z_0 - 3$

$PC = (x_1 = y_0 - z_0 \wedge x_1 = z_0 \wedge z_1 = z_0 - 3 \wedge 4 * z_1 < x_1 + y_0 \wedge 25 \leq x_1 + y_0)$

Check satisfiability with an SMT solver (e.g., `http://rise4fun.com/Z3`)

Symbolic execution today

- Leverages on the impressive advancements for SMT solvers
- Modern symbolic execution frameworks are not purely symbolic, and not necessarily static:
  - They can follow a concrete execution while collecting constraints along the way, or
  - They can treat some of the variables concretely, and some other symbolically
- This allows them to scale, to handle closed code or complex queries

Symbolic execution today

- C (actullay llvm) `http://klee.github.io/`
- Java (more than a symbolic executer)
  `http://babelfish.arc.nasa.gov/trac/jpf`
- C# (actually .net)
  `http://research.microsoft.com/en-us/projects/pex/`
- ...