Formal methods: Lecture 2
11.02.2016

Model Checking I:
TRANSITION SYSTEMS

Model Checking (MC) problem: intuition

- Correct design means that certain correctness properties must be satisfied by the system under development
- Correctness properties state what behaviours/features are correct and what are not in the system.
- To apply rigorous verification methods both
  - system description and
  - correctness properties description

  must be formalised
- System is described formally with its *model*
- Properties are specified formally as *logic expressions*.

Model Checking (formally)

- Satisfaction relation symbolically:

$$M \models \varphi ?$$

"Does model *M* satisfy logic expression φ *?*"

- Property φ is stated often in temporal logic.
- *M* is a state-transition system that models the behavior of the implementation to be verified.

Procedural view:

- Model checking is a method of model *M* state space exploration to determine if it satisfies the property φ.

Advantage of MC

- Fully automatic
- Diagnostic trace (counter example) generated by checker helps to analyze the source of the problem
- Good for bug-hunting, i.e. a “debugger” that does not require full execution of your program.

Modeling

How to get *M*?

1. By the process of abstraction:
   - Makes verification possible by retaining the part of the system that is relevant to modeling;
   - Should not discard too much so that the result lacks certainty, or too little so that the verification is not feasible;
   - Usually done by human (novel automatic model extraction techniques are gaining popularity).
2. By observation and learning (model construction by machine learning)

Choice of models

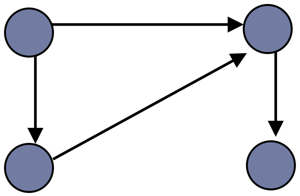
- We focus on state-transition systems. They are
  - acceptable by model checkers;
  - mostly finite set of states and transitions;
  - also push-down automata/systems are possible;
  - source programs can also be used as models, e.g., Pathfinder for Java code;
  - in symbolic encoding the logic formulae specify abstract properties instead of explicit state behavior modelling.

Modeling notions

- State
  - We want to express what is true in a particular state
  - A *state* is a “snapshot” of the system variables’ valuation(s).

- Transition represents relation between states.

  It can be an abstraction of

  - **C program** statement, e.g. *x*++;
  - an electronic circuit
  - or just an arrow, the source and destination states of which matter.

Atomicity

- Execution of a transition is atomic, i.e. uninterruptable once started.
- Atomicity determines the abstraction level of the model
  - too big step may miss intermediate states that are relevant;
  - too small step may blow up the model unnecessarily.
- Atomicity of transitions must alsoconsider concurrency
  - possible interleavings of transitions and interactions must be explicit.

Kripke Structure (*KS*)

One of the classical State Transition System models

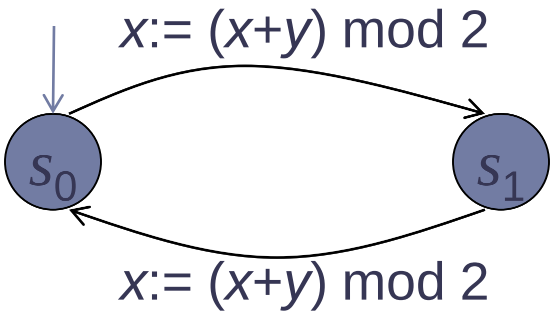
4-tuple (S, S_0, L, R) over a set of atomic propositions (AP) where

- S is a set of (control) states
- S_0 is initial state
- L is a labeling function: $S \rightarrow 2^{AP}$
- R is the transition relation: $S \times S$

Example of *KS*

Assume in s_0 x=1 and y=1

- $S = \{s_0, s_1\}$
- $S_0 = \{s_0\}$
- $R = \{(s_0, s_1), (s_1, s_0)\}$
- $L(s_0) = \{x=1, y=1\}$
- $L(s_1) = \{x=0, y=1\}$



Modeling Reactive Systems

- Reactive systems (RS) are STS that:
  - do not terminate;
  - interract with their environment constantly.

- Consider *KS* as a simple modeling language for RS-s
  - *though KS* is just one way of modeling them.

Properties of reactive systems to verify

- *race condition* - the output depends on the sequence of uncontrollable events. It becomes a *bug* when events do not happen in the order the programmer intended, e.g.
  - in file systems, programs may "collide" in their attempts to modify or access a file, which could result in data corruption;
  - in networking, two users of different servers on different ends of the network try to start the same-named channel at the same time.
- *deadlock* – all processes are waiting after each other infinitely for releasing the resources. Generally undecidable, practical decidability only for finite state processes.
- *starvation* - blocking resources for only some processes.
- etc.

Modeling Concurrent Programs with *KS*

- Steps of constructing KS from a program (by Manna, Pnueli):
  1. Abstract (sequential) component programs as logic relations.
  2. Compose the logic relations for the *concurrent program.*
  3. Compute a Kripke structure from the logic relations.

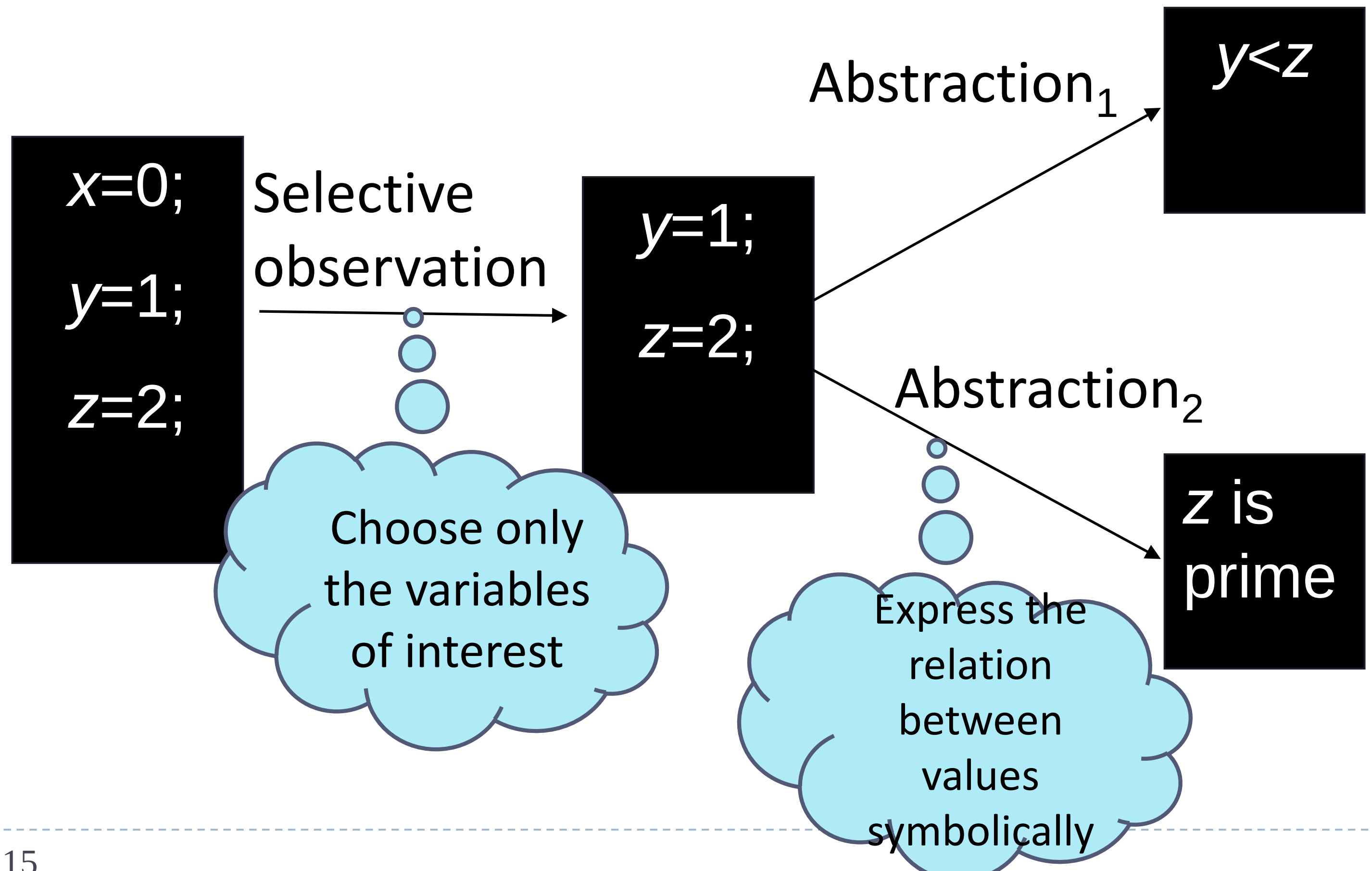
How does it work in practice?

Describing States

For abstracting states we use program variables and 1st order predicate logic…

- true, false, ¬, ∧, ∨, ∀, ∃, ⇒ extended with equality "=" and interpreted predicate symbols and function symbols:
  - even (x)
  - odd (x)
  - prime (x)
    - etc

Example of state abstraction steps



Representing States

- *Valuation* of a state
  - A mapping: $V \rightarrow \boldsymbol{V}$ from observable state variables V to their value domain $\boldsymbol{V}$.
- *Symbolic state* = set of explicit states
  - The set of states is described by a 1st order logic formula.
  - Instead of enumerating explicit states we use a logic formula describing the set $S_{0.}$
  - Example: $S_0 \equiv (x = 1) \wedge (y > 2)$

Representing a transition

- Transition abstracts a program command (or circuit)
  - Distinguish two sets of variables' values:

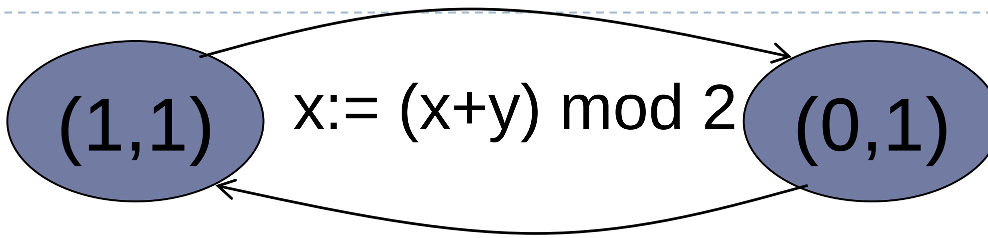
    V and *V'* for variable valuation in pre- and post-state of the transition, respectively
- Transition relation is a relation between *V* and *V'*
  - relation is expressable as a set of pairs of states
  - represented as a logic formula on *V*, *V '* with "=",
- Example:
  - Relation x' = x+1 describes the effect of program statement x:=x+1

From Logic Relation to Kripke Structure

Rules

- S (statespace) is the set of all valuations for V;
- S_0 is the set of all valuations that satisfy S_0 (a logic formula)
- If s and s' are two states, s.t. $(s,s') \in R(s,s')$ then the pair (s,s') is a transition in KS;
- L is defined so that $L(s)$ is the subset of all atomic propositions true in s.

Example



Explicit state KS:

- S_0 = {(1,1)}
- R = {((1,1), (0,1)), ((0,1),(1,1))}
- L(1,1) = {x=1, y=1}
- L(0,1) = {x=0, y=1}

- Symbolic state KS:
- $S_0 \equiv x = 1 \wedge y = 1$
- $R \equiv x' = (x+y) \bmod 2$
- $S = \mathbf{B} \times \mathbf{B}$, where $\mathbf{B} = \{0,1\}$

Abstracting parallel programs to KS

- A parallel program contains sequential processes
  - with synchronization primitives, e.g. wait, lock and unlock
  - processes may share variables
  - no assumption about the speed and execution order of these processes
- Program commands are labeled with $l_1 \ldots l_n$
- We use $C(l_1, P, l_2)$ to denote the logic relation of the transition that represents program P.

How to compute transition relation for sequential program fragments?

- Base case: atomic statements:
  - `skip` % has no effect on data variables
  - assignment: `x := e`

  Let C describe valuations before and after executing P: `x:=e`

$$C(l_1, \texttt{x:=e}, l_2) \equiv$$

$$pc = l_1 \wedge pc' = l_2 \wedge x' = e \wedge \mathit{same}(V \setminus \{x\})$$

  - $\mathit{same}(Y)$ means $y' = y$, for all $y \in Y$.

How to compute transition relation for sequential program fragments? (2)

- Sequential composition

  $C(l_0, P1 \; \mathtt{;} \; l: P2, l_1) = C(l_0, P1, l) \lor C(l, P2, l_1)$

- $C(l,$ `if b then` l_1: P1 `else` l_2: P2 `end if`, $l') =$

Conditional part

- $pc = l \wedge pc' = l_1 \wedge b \wedge \mathit{same}(V) \quad \lor$
- $pc = l \wedge pc' = l_2 \wedge \neg b \wedge \mathit{same}(V) \quad \lor$

Body part

- $C(l_1, P1, l') \lor C(l_2, P2, l')$

How to compute logic relations for concurrent programs?

Example: concurrent while-loops sharing a variable "turn"

```
L0: while (true) do
   NC0: wait (turn =0);
   CR0: turn := 1;
   end while
L0'
```

```
L1: while (true) do
   NC1: wait (turn =1);
   CR1: turn := 0;
   end while
L1'
```

- identify variables, including program counters;
- compute the set of states and set of initial states;
- compute transitions.

Example (continued I)

L0: while (true) do	L1: while (true) do
NC0: wait (turn =0);	NC1: wait (turn =1);
CR0: turn := 1;	CR1: turn := 0;
end while	end while
L0’	L1’

Identify variables, including program counters:

- V = { pc_0, pc_1, turn}
- domain of **pc_0** is L0, NC0, CR0, L0’
- domain of **turn** is {0,1}

Example (continued II)

```
L0: while (true) do
   NC0: wait (turn =0);
   CR0: turn := 1;
   end while
L0'
```

```
L1: while (true) do
   NC1: wait (turn =1);
   CR1: turn := 0;
   end while
L1'
```

- Compute the set of states and set of initial states
  - S= {(L0, L1, 1), (L0, L1, 0), (L0, NC1, 0), (L0, NC1, 1), …}
  - S_0 = {(L0, L1, 0), (L0, L1, 1)}

Example (continued III)

```
m: cobegin

L0: while (true) do
   NC0: wait (turn =0);
   CR0: turn := 1;
   end while
L0'

m': coend
```

- Compute transition relation separately & then compose them together:
  - For global program counter *dom*(pc) = {m, m', ⊥}
  - ⊥ represents that one of local processes is taking effect.

Example (continued IV)

```
m: cobegin

L0: while (true) do
    NC0: wait (turn =0);
    CR0: turn := 1;
    end while
L0'

L1: while (true) do
    NC1: wait (turn =1);
    CR1: turn := 0;
    end while
L1'

m': coend
```

- Transition relations of the composition:

$C(L0, P0, L0') \equiv turn' = turn+1 \wedge \textit{same}(V \setminus V0) \wedge \textit{same}(PC \setminus PC0)$

Summary

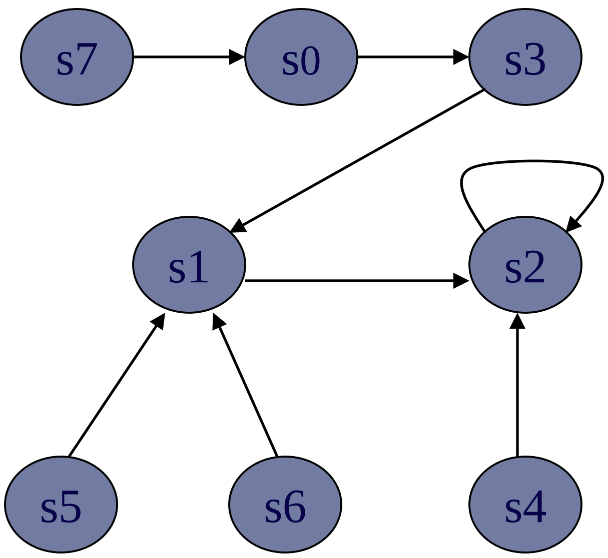
- We touched the concept of MC at very high level:
  - MC - an automatic procedure that verifies temporal and state properties
  - Requires input:
    - a state transition system
    - a temporal property
- State transition system – Kripke structure (KS):
  - KS structure is our (teaching) language
  - KS models reactive systems
- An example demonstrated how a concurrent program is translated to *KS*:
  - Step 1: Concurrent program is translated to logic relations
  - Srep 2: Logic relations are translated to *KS*.

Next lecture

- Temporal properties description logics
  - CTL*, CTL and LTL
  - Their semantics
- CTL model checking on Kripke structure

Exercise

- Give your explicit value definition to APs p, q, r.



L(s0) = {¬ p, ¬ q, ¬ r}

L(s1) = {¬ p, ¬ q, r}

L(s2) = {¬ p, q, ¬ r}

L(s3) = {¬ p, q, r}

L(s4) = {p, ¬ q, ¬ r}

L(s5) = {p, ¬ q, r}

L(s6) = {p, q, ¬ r}

L(s7) = {p, q, r}