ITI8531: Lecture 2
Module I: Model Checking
Topic: State transition systems

J.Vain

08.02.2017

Model Checking (MC) problem: intuition

- *Correct design* means that the system under development must satisfy design requirements. The requirements are stated as correctness properties
- Correctness properties state what behaviours/features are correct and what are not in the system.
- To apply rigorous *verification methods* formalization is needed:
  - system description
  - correctness properties
- System is described formally with its *model*
- Properties are specified formally by *assertions expressed in logic*

Model Checking (formally)

- <u>Satisfaction relation</u> (symbolically):

$$M \models \varphi\ ?$$

"Does model M satisfy logic assertion φ ?"

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

<u>Procedural definition</u>:

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

Why MC?

- MC is fully automatic
- Good for bug-hunting because the “debugger” i.e. model checker that does not require full execution of your program
- Traceability – the diagnostic trace (counter example) generated by model checker helps in analyzing and detecting the sources of design bugs.

Modelling

Where the model *M* comes from?

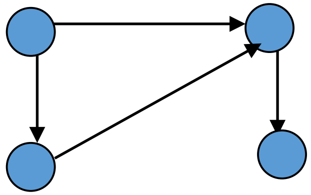
1. Formal modelling
   - It is a process of abstraction
   - It makes verification possible by retaining the part of the system that is relevant to modeling
   - It should not discard too much so that the result lacks certainty, or
   - discard too little so that the verification is not feasible
2. Modelling techniques:
   - “manual“ composition by applying model patterns, abstractions, domain knowledge,…
   - automatic modelling by applying machine learning methods:
     - state and/or IO monitoring and automata learning from these logs
     - model extraction from code.

Choosing the modelling formalism?

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

Modelling notions

- State
  - We want to express what is true in a particular state
  - A ***state*** is a “snapshot” of the system variables’ valuation(s), e.g.
    - if a system is described by variables *x*, *y*, *z* then valuation *x*=2.4, *y*= 3.14, *z*=10 is one of its possible states.



- Transition represents relation between states.

  It can be an abstraction of
  - **C program** statement, e.g. *x*++ transforming state where *x*=12 to a new state where *x*=13;
  - an electronic circuit;
  - or just an arrow, the source and destination states of which matter.

Atomicity of state transitions

- 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 important;
  - too small step may blow up the model unnecessarily.
- Atomicity of transitions must also consider concurrency
  - possible interleavings of transitions and interactions of parallel transitions systems must be explicit in the model.

Kripke Structure (*KS*)

KS is one of the classic State Transition System models

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

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

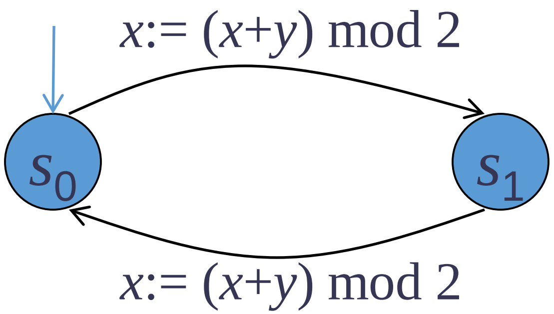
<u>*Note*</u>:

L specifies what conditions the explicit states of a symbolic state have to satisfy.

Example of *KS*

Assume the state vector consists of 2 state variables x and y

- Initially 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 (in general);
  - repeatedly interract with their environment.

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

Some properties of RS to be verified

- *Race condition* - the output depends on the order of uncontrollable events. It becomes a *bug* when events do not happen in the order the programmer has intended, e.g.
  - in file systems, programs may be conflicting in their attempts to modify the file, which could result in data corruption;
  - in networking, two users of different servers at different ends of the network try to start the same-named channel at the same time.
- *Deadlock* – all processes are infinitely waiting after each other for releasing the resources. Generally undecidable, practical decidability is granted only for finite state processes.
- *Starvation* - some processes are blocked from resources.
- etc.

Modeling Concurrent Programs with *KS*

How to construct KS from a (parallel) program?

Approach by by Manna, Pnueli:

1. Abstract the sequential components of the program as logic relations.
2. Compose the logic relations for the full *concurrent program.*
3. Compute a Kripke structure from these logic relations.

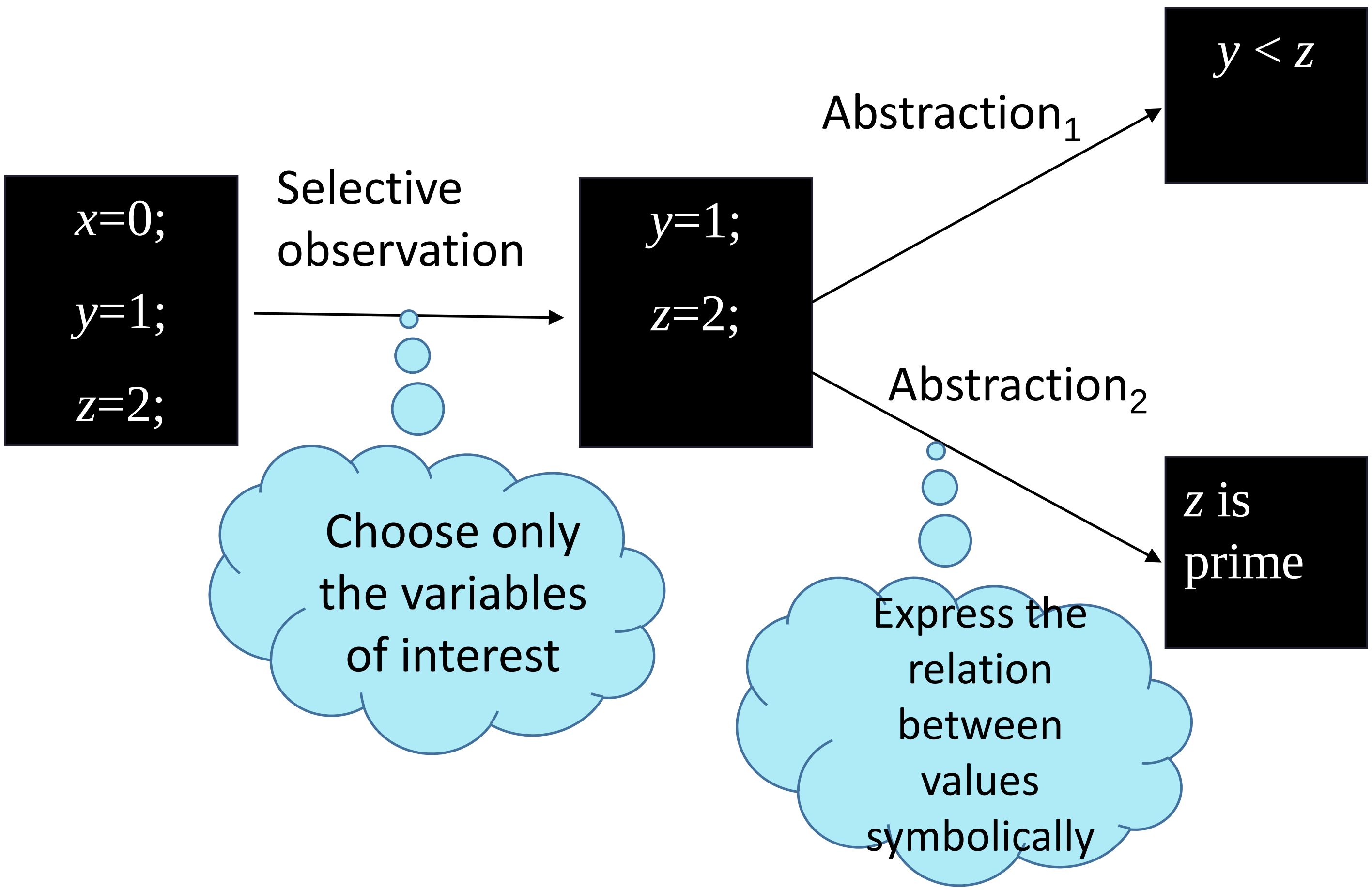
Let us look how it works on an example?

Describing States

- For abstracting states we use program variables and 1st order predicate logic…
- In the logic we have
  - true, false, $\neg$, $\wedge$, $\vee$, $\forall$, $\exists$, $\Rightarrow$
  - equality "="
  - interpreted predicate and function symbols:
    - $even(x)$
    - $odd(x)$
    - $prime(x)$

    …

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
  - Instead of enumerating explicit states we use a constraint that describes the set.
  - This constraint is a 1st order logic formula.
  - Example: $S_i \equiv (x = 1) \wedge (y > 2)$

Representing a transition

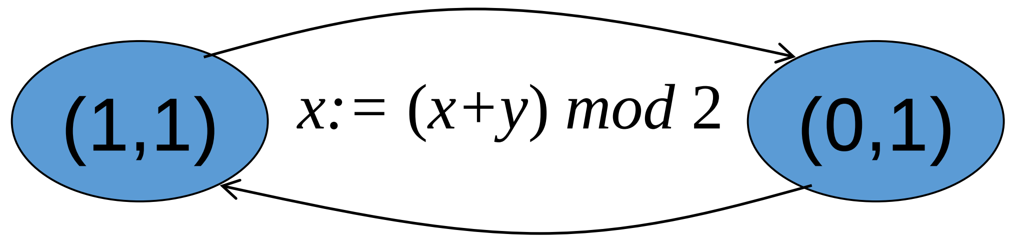
- A transition abstracts a program command
  - We need to distinguish two sets of variables' values:
    V and V' for variable valuation in pre- and post-state of the transition, respectively
- Transition relation is relation between V and V'
  - relation is expressable as a set of pairs of states
  - represented as a boolean equation on V, V'
- 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:
  - $S_0 \equiv x = 1 \wedge y = 1$
  - $R \equiv x' = (x+y) \text{ mod } 2$
  - $S = \mathbb{B} \times \mathbb{B}$, where $\mathbb{B} = \{0,1\}$

Abstracting parallel programs to KS

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

How to compute the transition relation for sequential components? (1)

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

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

$$C(l_1, \texttt{x:=e}, l_2) \equiv pc = l_1 \wedge pc' = l_2 \wedge x' = e \wedge same(V \setminus \{x\})$$

where

$same(Y)$ means $y' = y$, for all $y \in Y$.

How to compute the transition relation for sequential components? (2)

- Sequential composition

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

- If-command

$C(l, \texttt{if b then}\ l_1\!: \mathrm{P1}\ \texttt{else}\ l_2\!: \mathrm{P2}\ \texttt{end if}, l') =$

Conditional part:

$\mathrm{pc} = l \land \mathrm{pc}' = l_1 \land \mathrm{b} \land \mathit{same}(V) \quad \lor$

$\mathrm{pc} = l \land \mathrm{pc}' = l_2 \land \neg\, \mathrm{b} \land \mathit{same}(V) \quad \lor$

Body part:

$C(l_1, \mathrm{P1}, l') \quad \lor$

$C(l_2, \mathrm{P2}, l')$

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
   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:

- V = { `pc_0, pc_1, turn`}
- dom (`pc_0`) = { `L0, NC0, CR0, L0'`}
- dom(`turn`)= $\{0, 1\}$

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
   C0:wait(turn=0);
   CR0:turn:=1;
   end while
L0’
```

m’: coend

- Compute transition relations for processes separately
- Concatenate state vectors and compose transition relations together:
  - For global program counter dom(pc) = {m, m’, ⊥}
  - ⊥ represents that one of the local processes is taking effect, which one we don’t care.

Example (continued IV)

m: cobegin

```
L0: while(true) do
   C0: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:

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

Summary

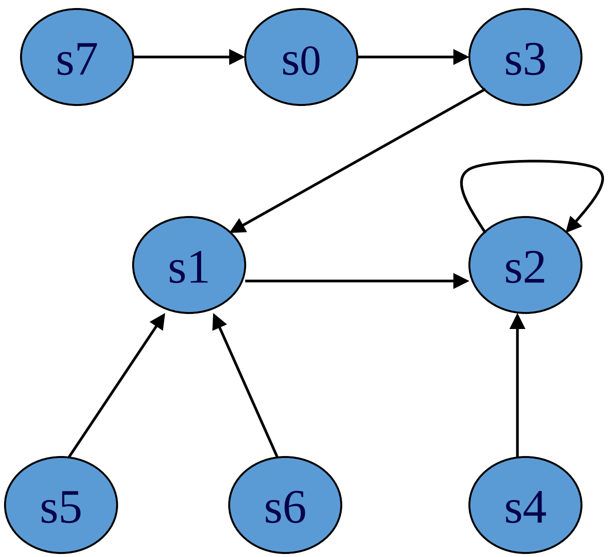
- We touched the concept of MC at very high level:
  - MC is an automatic procedure that verifies temporal and state properties
  - Requires input:
    - a state transition system
    - a temporal property
- State transition system – Kripke structure (KS):
  - KS structure is our (teaching) modelling 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 algorithms 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}