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

Jüri Vain

03.02.2022

Model Checking (MC) problem: intuition

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

Advantages of MC?

- Model checkers do not require full execution of programs, they run on program's abstract representation.
- MC is *fully automatic*
- Large number of tools (Spin, Java Pathfinder, …), see *https*://en.wikipedia.org/wiki/List_of_model_checking_tools
- MC is good for *bug-hunting* because the "debugger" is native component of each model checker.
- *Traceability* – the diagnostic trace (counter example) generated by model checker helps in analyzing and detecting the sources of design bugs.

Model Checking (formally)

- Satisfaction relation (symbolically):

$$M \models \varphi\ ?$$

"Does model M satisfy logic assertion φ?"

- Behavioural property is expressed as *temporal logic formula* φ.
- Model M is a state-transition system that *formalizes the behavior* of the system to be verified.

Procedural definition:

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

Modelling

How do we get the models?

1. Formal modelling
   - is a process of abstraction, i.e.,
   - it makes verification possible by retaining the part of the system that is relevant to properties of interest
   - should not discard too much so that the result lacks certainty, or
   - should not discard too little to avoid too complex verification tasks.
2. Modelling techniques:
   - "**manual**" construction by applying model patterns, abstraction, domain knowledge,…
   - **automatic** modelling:
     - by monitoring states and events, and applying ML methods on logs
     - model extraction from program code by parsing
     - extraction from (structured) natural language patterns

How to choose the modelling formalism?

- Hundreds of modelling languages, e.g. UML, SML, B, Z, …
- We focus on those which semantic bases is state-transition systems (STS).
- STS
  - are generally relevant for model checking;
  - represent finite set of states and transitions between states;
  - allow *abstraction*, i.e. symbolic encodings (logic formulae) specify abstract properties and relations instead of explicit states and transitions

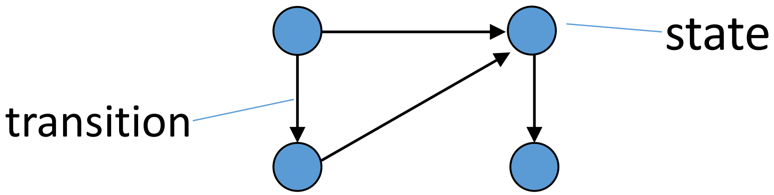
  Examples
  - push-down automata/systems are possible;
  - also source code can be used as model, e.g., Pathfinder uses Java code;

Modelling notions STS

- State
  - A *state* is a "**snapshot**" of the system variables' valuation
    - Example:

    Let *x*, *y*, *z* be state variables, then valuation *x*=2.4, *y*= 3.14, *z*=10 is one of its possible states.

    Graphically:



- Transition represents relation between states.

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

Atomicity of state transitions

- Execution of a transition STS is assumed to be *atomic*, i.e. *uninterruptable* once started.
- Atomicity of transitions determines the abstraction level of the model
  - too big state changing steps may miss intermediate states that are important;
  - too small steps may blow up the model unnecessarily.
- Atomicity of transitions must also consider *concurrency*, i.e.
  - possible interleavings of *transitions* and *interactions* of parallel transitions must be explicit in the models of paralleel systems.

Kripke Structure (*KS*)

KS is one of the classical State Transition Systems modelling formalisms

KS is a 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: $R \subseteq S \text{ x } S$

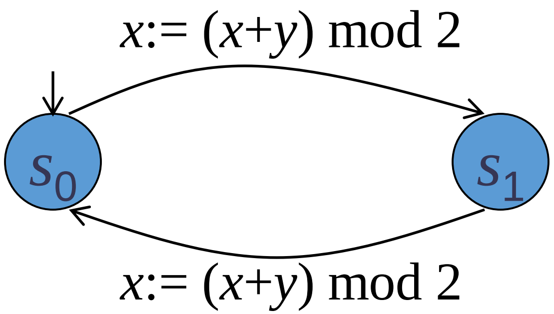
__*Note*__:

L specifies conditions the explicit states have to satisfy in the symbolic state.

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 system (RS) models are STS that:
  - do not terminate (in general);
  - interact repeatedly with their environment.

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

Properties: some examples of RS properties 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 systems.
- *Starvation* - some processes are blocked from some resources (also called, processes conspiracy against others).
- etc.

Modeling Concurrent Programs with *KS*

How to construct a KS of a (parallel) program?

Approach by Z.Manna, A.Pnueli:

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

Look how it works on an example?

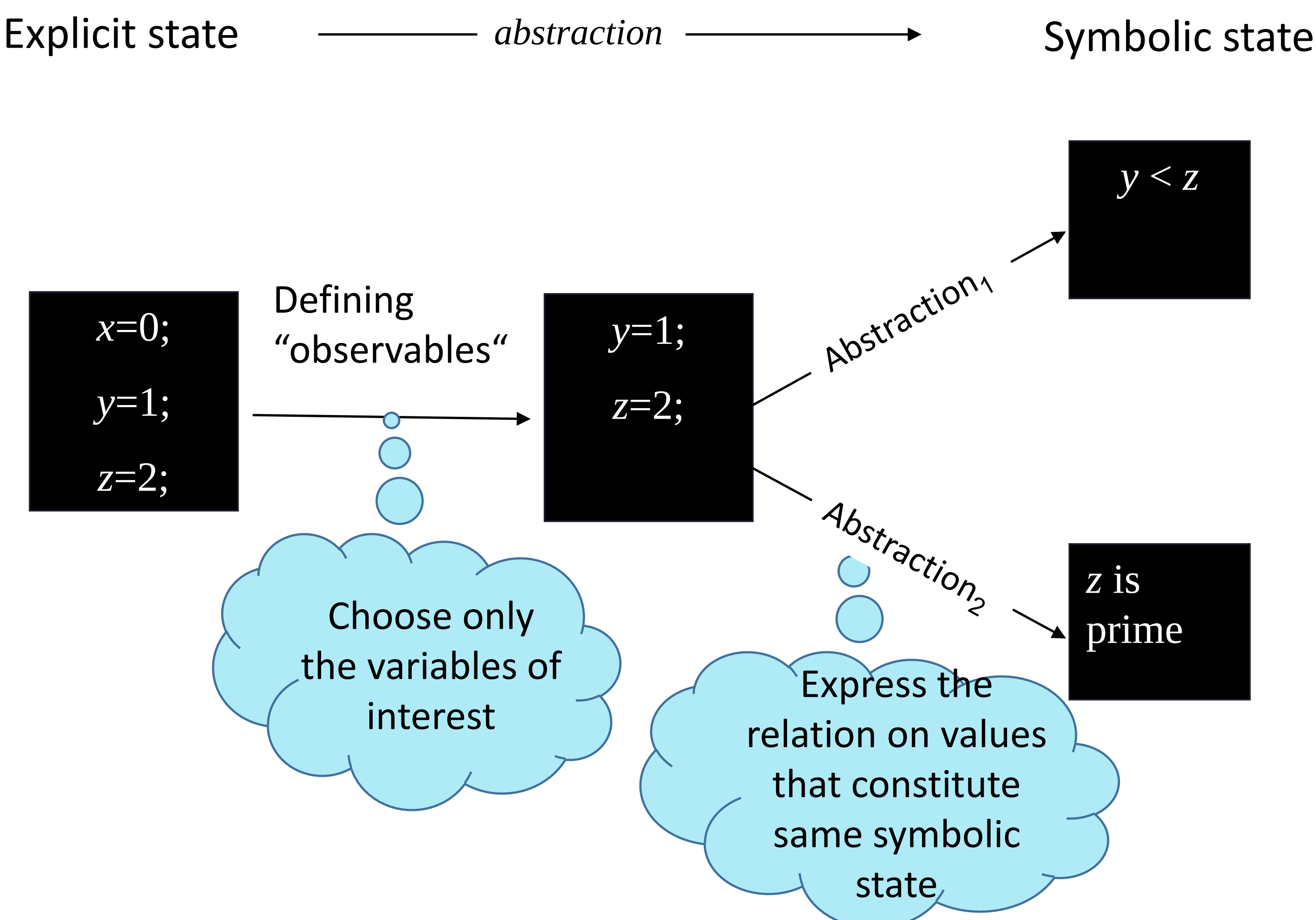
Step 1: abstracting sequential components
Step 1.1: Describing abstract states

- For abstracting states we use program variables and 1st order predicate logic (FOL)
- In the logic language we have symbols for
  - logic connectives: true, false, $\neg$, $\wedge$, $\vee$, $\forall$, $\exists$, $\Rightarrow$
  - arithmetic predicates: $=$, $>$, $<$,
  - other interpreted predicates and functions:
    - $even(x)$
    - $odd(x)$
    - $prime(x)$

    …

- NB! FOL does not have predicate variables

Example of state abstraction steps



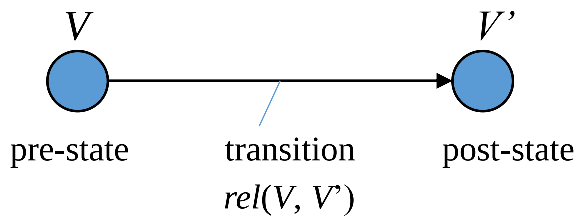
Representing States

- *Valuation* of a state
  - is a mapping: $V \rightarrow \boldsymbol{V}$ from observable state variables V to their value domains $\boldsymbol{V}$.
- *Symbolic state* represents not a single variable valuation but a **set of** them (explicit states)
  - Instead of enumerating explicit states in a symbolic state we use a constraint that describes the set of explicit values.
  - This constraint is a FOL formula.
  - Example: $S_i \equiv (x = 1) \wedge (y > 2)$

    Here all explicit states where x=1 and $y > 2$ constitute **one** symbolic state.

Step 1.2: Representing a transition

- A KS transition abstracts e.g. an execution of a program command
  - We 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' expressable as
  - a set of pairs of states
  - a boolean equation on V, V'
- Example:
  - Relation $x' = x+1$ describes the effect of program statement $x{:=}x+1$



Step 3: From Logic Relations to Kripke Structure (sequential systems case)

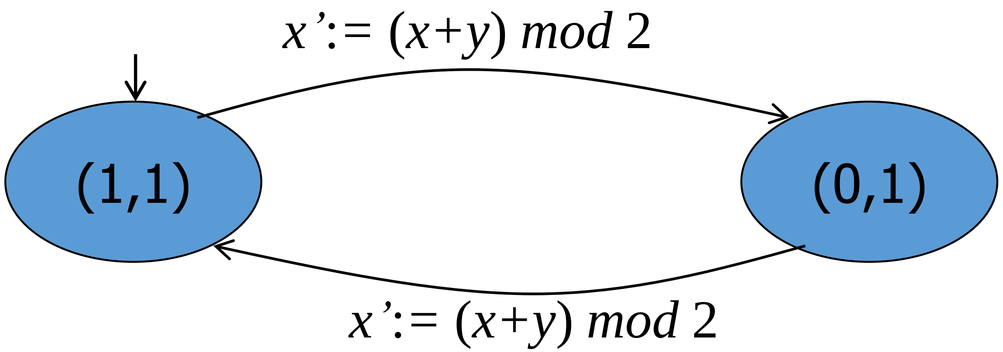
- Assume we have now FOL formulas describing states and state transitions of a sequential programm.

  - S - (explicit) statespace is a set of all valuations for V, e.g.

  if $V = \{v_1, \ldots v_n\}$ then $S = dom(v_1) \times \ldots \times dom(v_n)$

  - S_0 is the set of all valuations that satisfy $\mathcal{S}_0$ (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:

- State vector - (x, y)
- $S_0 = \{(1,1)\}$
- $L(1,1) = \{x{=}1, y{=}1\}$
- $L(0,1) = \{x{=}0, y{=}1\}$
- $R = \{((1,1), (0,1)), ((0,1),(1,1))\}$

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

Step 2: Abstracting parallel programs

- A parallel program consists of sequential processes
- Sequential processes
  - are composed of commands, e.g. `skip,:=,if,while`, …
  - are synchronized with primitives, e.g. `wait, lock` and `unlock`
  - may share variables
- In untimed models there is no assumption about the speed and execution order of processes (maximum concurrency).
- Program commands are labeled with labels $l_1, \dots, l_n$
- We use $C(l_1, P, l_2)$ to denote the logic relation of the state transition implemented by programm *P* that starts in state l_1 and terminates in state l_2.

Step 2.1: Constructing transition relation of processes? (1)

- Base case: atomic commands, e.g. `skip` and “`:=`” :
  - `skip` has no effect on data variables
  - assignment: `x := e`

Let C describe relation between valuations of variables before and after executing program P (label l_1 denotes pre-state and l_2 post-state of P)

If $P \equiv$ `x:=e` % includes only assignment

then

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

set difference

where

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

pc - program counter

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

- Sequential composition of programs P1 and P2

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

- If-command (l_1 and l_2 label then and else brances respectively)

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

Conditional part:

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

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

Body part:

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

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

How to compute logic relations for paralleel processes?

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'
```

- Notations: NC and CR label non-critical and critical region of the processes.
- Abstraction process:
  1. identify variables, including program counters pc0 and pc1;
  2. compute the set of states and set of initial states;
  3. compute transitions;
  4. aggregate processes.

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'
```

1. 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'
```

2. Compute the set of states and set of initial states

   State vector: (pc0, pc1, turn)

   State space: $S = \{(\mathrm{L0, L1, 1}), (\mathrm{L0, L1, 0}), (\mathrm{L0, NC1, 0}), (\mathrm{L0, NC1, 1}), \ldots\}$

   Inital states: $S_0 = \{(\mathrm{L0, L1, 0}), (\mathrm{L0, L1, 1})\}$

Example (continued III)

m: cobegin

```
L0: while(true) do
   C0:wait(turn=0);
   CR0:turn:=1;
   end while
L0'
```

||

m': coend

3. Compute transition relations for processes separately
4. 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:
  - e.g. move of the process $P0$

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

Summary

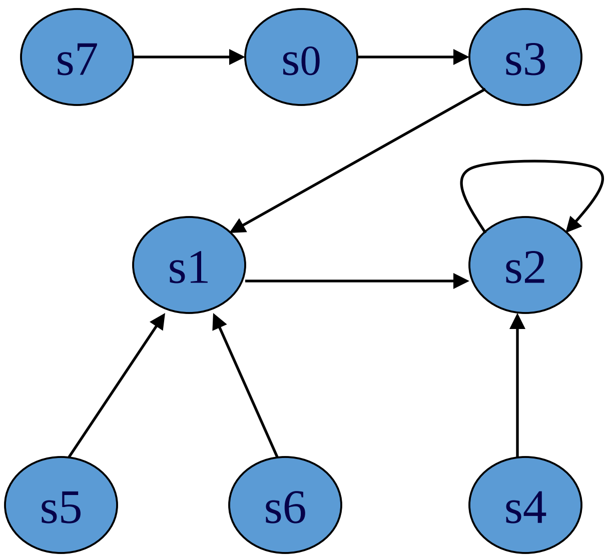
- We touched the concept of MC at very high level:
  - MC is an automatic procedure that verifies temporal and state properties of systems by exploring their models state space.
  - MC requires input:
    - a state transition system
    - a temporal property
- State transition system – Kripke structure (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* (topic of next lecture).

Next lecture

- Temporal logics for property description
  - CTL*, CTL and LTL
  - Their semantics
- CTL model checking algorithms for Kripke structure

Exercise

- Given a KS with labeling function L on boolean variables p, q, r
- Specify transition relation between states symbolically:



$L(\mathrm{s0}) = \{\neg p, \neg q, \neg r\}$
$L(\mathrm{s1}) = \{\neg p, \neg q, r\}$
$L(\mathrm{s2}) = \{\neg p, q, \neg r\}$
$L(\mathrm{s3}) = \{\neg p, q, r\}$
$L(\mathrm{s4}) = \{p, \neg q, \neg r\}$
$L(\mathrm{s5}) = \{p, \neg q, r\}$
$L(\mathrm{s6}) = \{p, q, \neg r\}$
$L(\mathrm{s7}) = \{p, q, r\}$

Transition relation $R \equiv \bigvee_i R_i$ *where* …, $R_{0,3} \equiv same(p) \wedge \neg q \wedge \neg r \wedge q' \wedge r'$