Modelling systems, model extraction, and basic model checking
[Automated Reasoning, 2012/2013 1b — Lecture 2]

Doina Bucur

RuG

Nov 2012
In this lecture...

Modelling computing systems
  Reactive systems
  Kripke structures. Definition and intuition

Extracting models from hardware, software, and protocols
  Digital circuits: synchronous and asynchronous
  Software: sequential and concurrent
  Communication protocols
  Modelling with Promela

Historical and bibliographical notes

Assignment

We show how to be a model checker yourselves. You learn to model various systems (in a formalism of states and transitions called transition system), and to extract one from code. The checking of the model against a state property is immediate: you simply browse through the state space of the model.

We apply this manual model checking to mutual exclusion algorithms. The assignments then move you towards using Spin to automatize this.
Systems may be categorized:

- In what regards **implementation**: software, hardware (logic circuits, etc.); this may generalize to systems other than computing systems.

- In what regards system **behaviour** and its complexity:
  - deterministic or non-deterministic
  - terminating or not terminating
  - synchronous or asynchronous
  - “lazy” or real-time
  - sequential or concurrent
  - transformational or reactive.
Reactive systems

A *transformational* computation is not interruptible (below left, represented as an input/output black box).

A reactive system *interacts* with the environment; input events may arrive at all times (right, as a “black cactus”).

Examples: device drivers, networked machines, chips on circuit boards, social humans, etc.; i.e., most *complex* systems.

They may be any combination of (non)terminating, (not) concurrent, (a)synchronous, etc.
Modelling a reactive system

...is by writing:

- Its internal **state** at any given time; this state describes current values stored in variables, program counter, heap, registers.
- What this state will **transition** into, given an internal or external event.

For this, it is standard to use a variation of finite-state machines called **Kripke structures**.
Kripke structures

Definition (Kripke structure)

A Kripke structure $M$ over a set of atomic propositions $AP$ is a tuple $M = (S, S_0, T, L)$ where

- $S$ is a finite set of states, or state space;
- $S_0 \subseteq S$ is the subset of initial states;
- $T \subseteq S \times S$ is a transition relation which is total: $\forall s \in S, \exists s' \in S$ so that $T(s, s')$;
- $L : S \rightarrow \mathcal{P}(AP)$ is a function labelling each state with that subset of atomic propositions which are true in that state.
A simple example: from code to Kripke structure

```c
/* x and y start at 0 */
11 while (true)
12 x = (x+1) mod 3, y = x+1;
```

- State $s \in S$ is a valuation of $V = \{x, y\}$;
- With $D$ the variables’ domain, $S \subseteq D \times D$; only reachable states are shown here;
- In $s_0$, $x = 0$, $y = 0$;
- $AP$ expresses exactly these valuations;
- A transition $\alpha \in T$ encodes an atomic assignment.
Some intuition behind this formalism

S must be finite. This is not an artificial limitation of the model: in all information systems, $D$ may be large (think 8-bit registers, C data types), but is finite. Digital systems never encode full $\mathbb{N}$, $\mathbb{Z}$, $\mathbb{R}$.

For a correct model, transitions must represent atomic system statements. Deciding on atomicity takes language, compiler, OS scheduler knowledge:

- Basic assignments in high-level languages may be atomic.
- However, machine-language translations may have finer grain.
- Atomic \{code\} constructs, if available, make code atomic.
- Disabling interrupts in sequential code makes it atomic.

$T$ having to be total is an artificial (but harmless) feature of the model.
...and some intuition behind model checking

In that simple example: suppose you want to know whether

- y ever becomes equal to 3
- x and y can be equal to 1 coincidently

You can answer all these by searching through the model. This is what a model checker also does.
An equivalent logic representation for Kripke structures

As per definition, the information is in **state labels**:

- $s_0 \equiv \{x=0, y=0\}$
- $s_1 \equiv \{x=1, y=2\}$
- $s_2 \equiv \{x=2, y=3\}$
- $s_3 \equiv \{x=0, y=1\}$

In this equivalent predicate-logic form, the same information is on **transitions** instead:

- A state is a conjunctive formula: $s_0 \equiv (x = 0) \land (y = 0)$.
- Transitions, also:
  $\alpha_i \equiv (x' = (x + 1) \mod 3) \land (y' = (x' + 1)), \forall i = 0..3$.
  Primed variables represent **next-state variables**.
- Close to program execution.
- A next state $s_1$ is the conjunction of the previous state $s_0$ and the transition $\alpha_0$. 

> A state is a conjunctive formula: $s_0 \equiv (x = 0) \land (y = 0)$.

> Transitions, also:

  $\alpha_i \equiv (x' = (x + 1) \mod 3) \land (y' = (x' + 1)), \forall i = 0..3$.

Primed variables represent **next-state variables**.

> Close to program execution.

> A next state $s_1$ is the conjunction of the previous state $s_0$ and the transition $\alpha_0$. 

Paths in Kripke structures

A path $\pi$ from state $s$ in a Kripke structure is an infinite sequence of states

$$\pi = s_0 s_1 s_2 \ldots$$

so that $s_0 = s$ and $(s_i, s_{i+1}) \in T, \forall i \geq 0$.

$\pi^i$ is the suffix of $\pi$ starting at $s_i$.

Clearly, infinite paths exist in finite-state machines.

Later, we ask whether temporal properties: “only after $x$ becomes 1, $y$ can become greater than $x$”. For these, we will examine paths.

You can now imagine that the complexity of model checking will come with:

- the number of states in the model
- how expressive the property is
A synchronous circuit

These are called \textit{sequential} circuits: the output depends both on current and on past inputs (larger state space). They include memory (flip-flops): $Q$ becomes equal to $D$ only after a clock pulse.

- Three boolean variables, $D = \{0, 1\}$, $S = D \times D \times D$; say all start equal to 0.
- At each transition, the bits change by $b'_0 \leftrightarrow \neg b_0$, etc; the changes are \textit{synchronized};
- The one transition is the conjunction of these changes:

$$\left((b'_0 \leftrightarrow \neg b_0) \land (b'_1 \leftrightarrow b_0 \oplus b_1) \land (b'_2 \leftrightarrow (b_0 \land b_1) \oplus b_2)\right)$$

- Modulo-8 counter.
An asynchronous circuit

These are called *combinatorial* circuits: the output depends solely on the current input (no memory, no notion of time, no means of synchronization). They are constructed with logic gates.

- Three boolean inputs, \( V = \{ b_0, b_1, b_2, \} \), \( D = \{ 0, 1 \} \), \( S \subseteq D \times D \times D \);
- At each transition, the bits change independently by \( b'_0 \leftrightarrow \neg b_0 \), etc; these subtransitions may *interleave*;
- The transition *could* be written as a *disjunction* of these subtransitions:

\[
(b'_0 \leftrightarrow \neg b_0) \lor (b'_1 \leftrightarrow b_0 \oplus b_1) \lor (b'_2 \leftrightarrow (b_0 \land b_1) \oplus b_2)
\]

- Incrementing circuit.
Sequential software

We consider asynchronous software (a single processor). You can automatically translate a program into a transition system.

```c
/* x, y start at 0; z is not initialized */
while (true)
    x = (x+1) mod 3, y = x+1;
```

One way: Generate the control-flow graph from program source (any compiler also does that). Then, start with the initial Kripke state, and add a transition (and state) for each program statement.

```
0,0,*
1,2,*
2,3,*
0,1,*
```

(CFG)

(Kripke structure)
Sequential software

Another way: Start by giving the entry point to each basic program statement a label \( l \); the program counter will get these values:

```c
/* x, y start at 0; z is not initialized */
11 while (true)
12 x = (x+1) mod 3, y = x+1;
13
```

Then, translation \( Tr(l_1 P l_3) \) gives the transition relation for \( P \) in predicate-logic form:

- while loop: \( Tr(l_1 \ while(\text{true}) \ \{l_2 \ P_2\} \ l_3) \) is the disjunction
  \[
  (pc = l_1) \land (pc' = l_2) \land \text{same}(x, y, z) \lor Tr(l_2 \ P_2 \ l_1)
  \]

- assignment: \( Tr(l_2 \ x = (x+1) \mod 3, y = x+1; \ l_1) \) is the conjunction
  \[
  (pc = l_2) \land (x' = (x + 1) \mod 3) \land (y' = x' + 1) \land \text{same}(z) \land (pc' = l_1)
  \]
Generalizing \( \text{Tr}(S) \) somewhat

\[
\text{Tr}(l \ x=y; \ l') \equiv \\
(pc = l) \land (x' = y) \land (pc' = l') \land \text{same}(V \setminus \{x\})
\]

\[
\text{Tr}(l \ \text{if}(b) \{l_1 \ P_1\} \ \text{else} \{l_2 \ P_2\} \ l') \equiv \\
(pc = l) \land (b) \land (pc' = l_1) \land \text{same}(V) \lor \\
(pc = l) \land (\neg b) \land (pc' = l_2) \land \text{same}(V) \lor \\
\text{Tr}(l_1 \ P_1 \ l') \lor \\
\text{Tr}(l_2 \ P_2 \ l')
\]

\[
\text{Tr}(l \ \text{while}(b) \{l_1 \ P_1\} \ l') \equiv \\
(pc = l) \land (b) \land (pc' = l_1) \land \text{same}(V) \lor \\
(pc = l) \land (\neg b) \land (pc' = l') \land \text{same}(V) \lor \\
\text{Tr}(l_1 \ P_1 \ l)
\]

\[
\text{Tr}(l_1 \ P_1 \ l_2 \ P_2 \ l') \equiv \\
\text{Tr}(l_1 \ P_1 \ l_2) \lor \text{Tr}(l_2 \ P_2 \ l')
\]

This is what a model checker will do.
Concurrent software

Concurrency simply means interleaving the transitions of the sequential programs (as for asynchronous circuits). Processes share variables.

Take the concurrent program \( P \equiv I (P_1 \parallel P_2 \parallel \ldots \parallel P_n) I' \), where:

- The program counters \( pc_i \) form a set \( PC \).
- The sets of variables each process can access are \( V_1, V_2, \ldots V_n \); their union is \( V \).
- Add start/end labels \( l_i \) and \( l'_i \) for each process \( P_i \).

Essentially, \( Tr(I P I') \) is:

\[
\bigvee_{i=1}^{n} \left( Tr(l_i P_i l'_i) \land \text{same}(V \setminus V_i) \land \text{same}(PC \setminus pc_i) \right)
\]

(excluding a ‘first’ and ‘last’ formula starting and ending all processes)
A mutual exclusion example

Have $P_0 \parallel P_1$ symmetric processes, where $P_0$:

10 while (true)
nc0 wait_until(turn=0); /* busy waiting */
cr0 turn = 1;
10'

and symmetric for $P_1$ (any 0 becomes 1 and vice versa).
Consider both initial states.

$P_0$ is represented as predicate logic (below), then as Kripke structure (right).

$Tr(l_0 P_0 l'_0) \equiv$

$(pc_0 = l_0) \land (pc'_0 = nc_0) \land \text{same}(turn) \lor$
$(pc_0 = nc_0) \land (turn = 0) \land (pc'_0 = cr_0) \land \text{same}(turn) \lor$
$(pc_0 = nc_0) \land \neg(turn = 0) \land (pc'_0 = nc_0) \land \text{same}(turn) \lor$
$(pc_0 = cr_0) \land (turn' = 1) \land (pc'_0 = l_0)$
...Then, $P_0 \parallel P_1$ is:

Now check visually:

- Does this program ensure mutual exclusion?
- Can a process be denied forever entrance to its critical region? Why?
- If you find hard to answer the “Why?”, try labelling all your graph’s transitions, so that you can pinpoint transitions to the original code.
Communication with channels

For communication protocols, the model includes:

- Channels $c$ to send on and receive from; these are like shared variables (queues);
- Sending constant $m$ on $c$ is $c!m$; this is like an assignment $c = m$;
- Receiving variable $x$ on $c$ is $c?x$; this is also like an assignment $x = c$;
- Synchronous or asynchronous (i.e. buffered)!
Modelling with PROMELA

High-level language; features from the Communicating Sequential Processes (CSP, C. A. R. Hoare) logic, and C. You have:

- processes and typed data variables, including inter-process communication channels with sophisticated operations;
- control-flow similar to that of C;
- a form of macro called inline (no functions);
- channels may model either asynchronous or synchronous communication, and may include a buffer;
- a great way to model busy-waiting on a condition: \((a == b)\) is a statement blocking the process until \(a\) equals \(b\);
- ...and many quirky features.
Historical notes


Unfolding programs into **transition systems** has been used for verification of programs since the 70s, e.g. *Formal verification of Parallel Programs*, R. M. Keller, in Comm. ACM, 19(7), 1976.

For getting a grip with **SPIN**, go for Ch. 2 *Building Verification Models*, Ch. 3 *An Overview of PROMELA*, Ch. 18 *Overview of SPIN Options*, Ch. 19 *Overview of PAN Options*, from *The SPIN Model Checker*, G. J. Holzmann.

Alternatives: the **SPIN** manual


You may also read on the topic from Ch. 2, *Modelling Concurrent Systems*, from *Principles of Model Checking*, C. Baier, J.-P. Katoen.
Assignment


(10%) Go back to the mutual exclusion on slides 18-19. Answer the Why? question stated there.

(45%) Do the modelling for Peterson’s mutual exclusion algorithm (with two processes); you find it on Wikipedia. Visually check the Wikipedia statements under “Mutual exclusion” and “Bounded waiting”.

Finally, fire up SPIN, find an implementation of this (peterson.pml in SPIN’s sources already contains an assertion for mutual exclusion), and check it with an exhaustive search. Decode the output and compare the size of this state space with the one above. Then, prove that mutual exclusion is not insured without the second wait condition (on turn), and simulate the error trace. See http://www.pst.ifi.lmu.de/~hammer/statespaces/peterson/ for some graphical entertainment.
Assignment administration

Handing in:

▶ in lab session or electronically;
▶ either way, before Tue next week (Nov 27), 1pm.

Grading:

▶ this is worth 10% of your Practical grade
  (and 6% of the final grade).