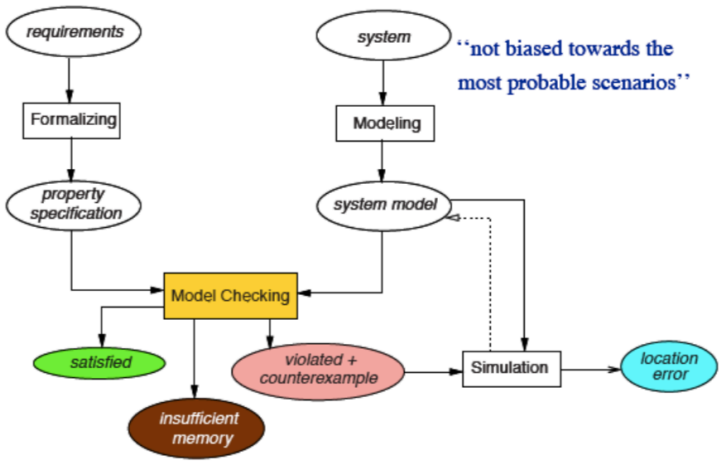
Conteúdo

1 Model Checking in a Nutshell

Model checking



- Gödel Prize 2000

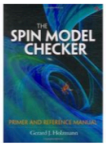
Moshe Vardi

Pierre Wolper

For work on model checking with finite automata

- ACM System Software Award 2001

Gerard J. Holzmann



SPIN book

- ACM Turing Award 2007

Edmund Clarke

E. Allen Emerson

Joseph Sifakis

> For their role in developing Model-Checking into a highly effective verification technology, widely adopted in the hardware and software industries

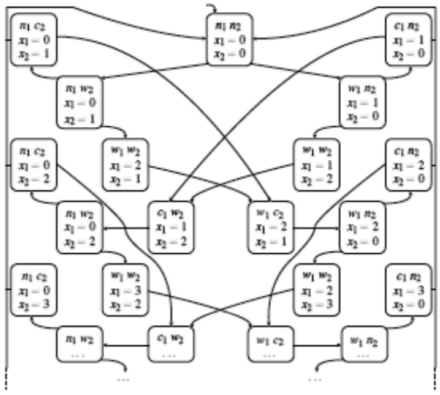
Model checking

- We start from a model not the real system
- Exhaustively checks the possible states of a model
- Reactive systems/concurrent are described by temporal properties
- A specification is verified if it is satisfied during the execution of the model

Model checking

Model checking is an automated technique that, given a finite-state model of a system and a formal property, systematically checks whether this property holds for (a given state in) that model.

What is a Model?



What is a Model?

Transition Systems/Automata

- States are labelled with basic propositions
- Transition relation between states
- Action-labelled transitions

Expressivity: all these are transition systems

- Programs
- Concurrent programs
- Hardward circuits
- Embebbed/Cyberphysical systems

What is a Property?

Examples

- Can the system enter in deadlock?
- Can two processes be simultaneously in a critical section?
- On termination, does the program provide the correct value?
- Can the system always reach a given state?

Temporal Logic

- Propositional logic
- Temporal operators such as *always* (G) and *eventually* (F)
- Interpreted over infinite state sequences (paths/linear)
- Or over (infinite) trees of states (branching)
- A propositional formula can be true in a state but not in another

Concurrency and atomicity

$$\begin{aligned} \textbf{proc}\ \mathsf{Inc} &= \textbf{while}\ \mathrm{true}\ \textbf{do}\ \textbf{if}\ x < 200\ \textbf{then}\ x := x + 1\ \textbf{fi}\ \textbf{od} \\ \textbf{proc}\ \mathsf{Dec} &= \textbf{while}\ \mathrm{true}\ \textbf{do}\ \textbf{if}\ x > 0\ \textbf{then}\ x := x - 1\ \textbf{fi}\ \textbf{od} \\ \textbf{proc}\ \mathsf{Reset} &= \textbf{while}\ \mathrm{true}\ \textbf{do}\ \textbf{if}\ x = 200\ \textbf{then}\ x := 0\ \textbf{fi}\ \textbf{od} \end{aligned}$$

Are the values of x always between 0 and 200? No, it depends on the scheduler:

- suppose $x = 200$
- the process Dec tests x and control goes to process Reset
- the process Reset tests x and does $x = 0$
- The control goes to process Dec and we have $x = -1$

Model checking

- Given a model $\mathcal{M}$ (*transition system*),
- a (initial) state s
- and a formula φ of a temporal logic (*specification*),
- the aim is
- $\mathcal{M}, s \models \varphi$
- i.e. φ is satisfied in the state s of $\mathcal{M}$
- a *model checker* is a program that decides this problem (i.e. either answers yes or no).

The Model Checking Process

Modeling: using a modeling language that allows the simulation of the model

Formalisation of the properties to be checked

Execution of a *model checker* that verifies if the property is satisfied in the model

Analysis:
- property is satisfied
- if the property *does not hold* a counterexample is produced and the model may be refined
- But the *model checker* can also be *out of memory*, as the number of states can be huge (*state-space explosion*)

Properties

***Reachability* (reachable states)** properties that ensure that a given state is reachable

Safety properties that has to hold in all the executions and all the states

Liveness some states must be eventually reached

Persistence in all executions from some state on the property has to hold

Fairness a given property must hold infinitely often. Example: no process is forgotten by the scheduler always

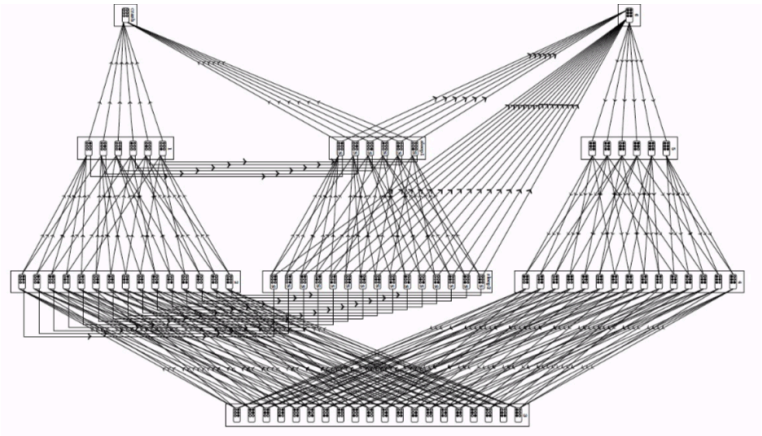
SWOT: Pros

- It is applicable to a wide range of applications such as embedded systems, software engineering, and hardware design.
- It supports partial verification, i.e., properties can be checked individually, thus allowing focus on the essential properties first. No complete requirement specification is needed.
- It is not vulnerable to the likelihood that an error is exposed; this contrasts with testing and simulation that are aimed at tracing the most probable defects.
- It provides diagnostic information in case a property is invalidated; this is very useful for debugging purposes.

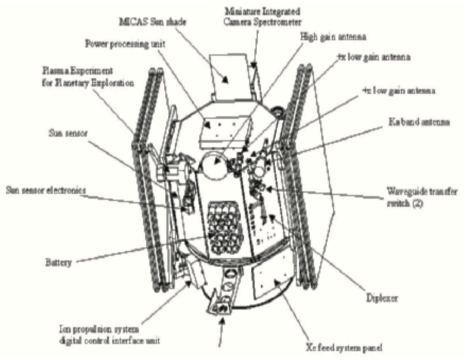
SWOT: Cons

- It is mainly appropriate to control-intensive applications and less suited for data-intensive applications as data typically ranges over infinite domains.
- It verifies a system model, and not the actual system (product or prototype) itself
- It checks only stated requirements, i.e., there is no guarantee of completeness.
- It suffers from the state-space explosion problem, i.e., the number of states needed to model the system accurately may easily exceed the amount of available computer memory.
- Its usage requires some expertise in finding appropriate abstractions

State explosion- Hubble telescope



Nasa Deep Space-1 Spacecraft (1998)



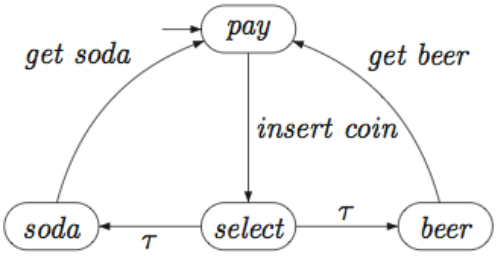
Several errors where discovered with Model checking (including one deadlock)

Other examples

- Security: Needham-Schroeder encryption protocol (error that remained undiscovered for 17 years unrevealed)
- Transportation systems (train model containing 10476 states)
- Model checkers for C, Java and C++ (used (and developed) by Microsoft, Digital, NASA)
- Successful application area: device drivers
- Dutch storm surge barrier in Nieuwe Waterweg (Rotterdam)
- Software in the space missiles
- NASA's Mars Curiosity Rover, Deep Space-1, Galileo
- LARS group@Jet Propulsion Lab
- Facebook @Monoidics (see Calcagno et al., Moving fast with software verification, 2014)

2 Transition Systems

Vending Machine



Transition System/Automaton/Model

A transition system is a tuple $T = (\mathbf{S}, Act, \longrightarrow, I, AP, \mathbf{L})$ where

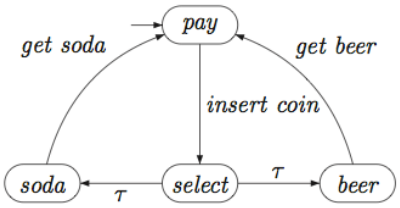
- S set of states
- Act set of actios
- $\longrightarrow \subseteq S \times Act \times S$ transition relation
- $I \subseteq S$ set of initial states
- AP set of proposicional variables (atomic propositions)
- $L : S \rightarrow 2^{AP}$ labelling function that associates a state with a set of propositions (that are true in that state)

Transition System/Automaton/Model

- if $(s, \alpha, s') \in \longrightarrow$ we write $s \xrightarrow{\alpha} s'$
- T may be nondeterministic
- given s, $L(s)$ gives the atomic propositions that are satisfied in s
- we can omit the actions Act if we are only interesed in the structure of the system (i.e. in reachable states)
- we can omit the initial states
- then a model is just

$$\mathcal{M} = (S, \longrightarrow, L)$$

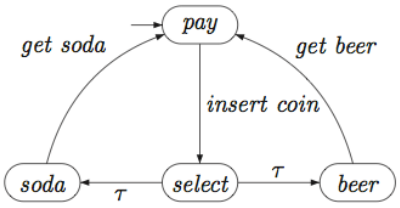
Vending machine - II



$$\begin{aligned} S &= \{pay, select, soda, beer\} \\ I &= \{pay\} \\ Act &= \{insert\_coin, get\_soda, get\_beer, \tau\} \\ \longrightarrow &= \{(pay, insert\_coin, select), (select, \tau, beer), \ldots\} \end{aligned}$$

τ describes actions that are not relevant for the model (e.g. internal actions)

Vending machine– III



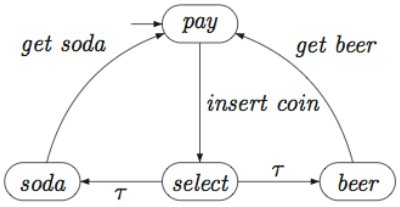
- We can assume that $AP = S$ and $L(s) = \{s\}$ for all $s \in S$
- Or, e.g., $AP = \{paid, drink\}$ and $L(pay) = \emptyset$, $L(select) = \{paid\}$, $L(soda) = L(beer) = \{paid, drink\}$

Successors and Predecessors

Let $T = (S, Act, \longrightarrow, I, AP, L)$. For $s \in S$ (or $C \subseteq S$) and $\alpha \in Act$, $Post$ is the set of direct α-sucessors and Pre of the predecessors:

$$\begin{aligned}
Post(s, \alpha) &= \{s' \in S \mid s \xrightarrow{\alpha} s'\} \\
Pre(s, \alpha) &= \{s' \in S \mid s' \xrightarrow{\alpha} s\} \\
Post(C, \alpha) &= \bigcup_{s \in C} Post(s, \alpha) \\
Pre(C, \alpha) &= \bigcup_{s \in C} Pre(s, \alpha) \\
Post(s) &= \bigcup_{\alpha \in Act} Post(s, \alpha) \\
Pre(s) &= \bigcup_{\alpha \in Act} Pre(s, \alpha) \\
Post(C) &= \bigcup_{\alpha \in Act} Post(C, \alpha) \\
Pre(C) &= \bigcup_{\alpha \in Act} Pre(C, \alpha)
\end{aligned}$$

Vending Machine III'



Compute $Post(select)$ and $Post(Pre(pay))$. $Post(select) = \{beer, soda\}$ and $Post(Pre(pay)) = Post(\{beer, soda\}) = \{pay\}$.

Terminal State

A state $s \in S$ is *terminal* iff $Post(s) = \emptyset$.

- For a transition system modeling a sequential computer program, terminal states occur as a natural phenomenon representing the termination of the program.
- For reactive systems we usually assume that they do not have terminal states or if they have it is not a desire property (deadlock).

Execution Fragment

Let $T = (S, Act, \longrightarrow, I, AP, L)$.

- A *finite execution fragment* is a alternating sequence of states and actions ρ such that
$$\rho = s_0\alpha_1 s_1\alpha_2 s_2 \ldots \alpha_n s_n,$$
and $s_i \xrightarrow{\alpha_{i+1}} s_{i+1}$ for all $0 \leq i \leq n$, $n \geq 0$. I.e.
$$\rho = s_0 \xrightarrow{\alpha_1} s_1 \xrightarrow{\alpha_2} s_2 \cdots \xrightarrow{\alpha_n} s_n$$

- The *execution fragment* is infinite if it is a infinite sequence
$$\rho = s_0\alpha_1 s_1\alpha_2 s_2 \ldots$$
i.e.
$$\rho = s_0 \xrightarrow{\alpha_1} s_1 \xrightarrow{\alpha_2} s_2 \cdots$$

Execution

A execution (fragment) is

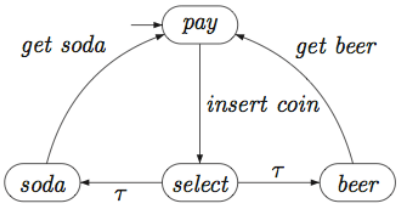
- *maximal* if finite and ends in a terminal state; or if infinite
- *initial* it starts in an initial state i.e. $s_0 \in I$

An *execution* is a fragment that is initial and maximal

Note: in general we will omit the actions α_i (and denote it a *path*)

$$\pi = s_0 \longrightarrow s_1 \longrightarrow s_2 \cdots$$

Vending Machine – IV



$$
\begin{aligned}
\rho_1 &= pay \xrightarrow{coin} select \xrightarrow{\tau} soda \xrightarrow{sget} pay \xrightarrow{coin} select \xrightarrow{\tau} soda \cdots \\
\rho_2 &= select \xrightarrow{\tau} soda \xrightarrow{sget} pay \xrightarrow{coin} select \xrightarrow{\tau} soda \cdots \\
\rho_3 &= pay \xrightarrow{coin} select \xrightarrow{\tau} soda \xrightarrow{sget} pay \xrightarrow{coin} select \xrightarrow{\tau} soda.
\end{aligned}
$$

ρ_1 is an execution but ρ_3 is not because there is no terminal state; neither is ρ_2 as it is not initial.

Reachable States

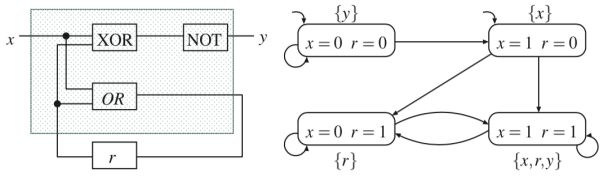
Let $T = (S, Act, \longrightarrow, I, AP, L)$. A state $s \in S$ is *reachable* if there exists a initial and finite fragment execution that ends in s

$$\rho = s_0 \xrightarrow{\alpha_1} s_1 \xrightarrow{\alpha_2} s_2 \cdots \xrightarrow{\alpha_n} s_n = s$$

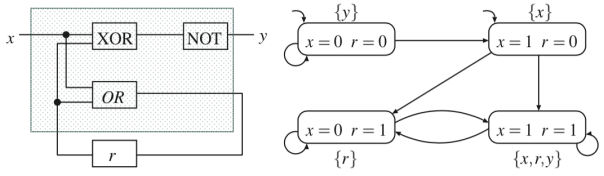
$Reach(T)$ is the set of all reachable states in T.

Sequential Hardware Circuits

A sequential circuit with one bit register r, 1 bit input x and 1 bit output y where the control function for y is $\lambda_y = \neg(x \oplus r)$ and the update function for r is $\delta_r = x \vee r$.



Sequential Hardware Circuits



- Suppose the initial value $r = 0$,
- $S = Eval(x, r)$ is the set of possible values of x and r
- $I = \{\langle x = 0, r = 0\rangle, \langle x = 1, r = 0\rangle\}$
- Actions are irrelevant

- Transitions result from the evaluation of λ_y e δ_r. For instance
- $\langle x = 0, r = 1\rangle \longrightarrow \langle x = 1, r = 1\rangle$ if the new input bit 1.
- $AP = \{x, y, z\}$ and each state is labelled with the variables that are true in that state. E.g. $L(\langle x = 0, r = 1\rangle) = \{r\}$ and $L(\langle x = 1, r = 1\rangle) = \{x, y, r\}$
- We can express the property:"the output bit y is 1 infinitely often"

Every sequential circuit can be represented by a transition system.

Sequential Circuits (general)

Consider a sequential circuit with n input bits $x_1, \ldots, x_n$, $y_1, \ldots, y_m$ output bits and $r_1, \ldots, r_k$ one bit registers. The input bits $x_1, \ldots x_n$ have nondeterministic values in $\{0, 1\}$ and the registers r_j have initial values $c_{0,j} \in \{0, 1\}$ for $1 \leq j \leq k$. The transition system that represents this circuit is $TS = (S, Act, \longrightarrow, I, AP, L)$ such that

- $S = Eval(x_1, \ldots, x_n, r_1, \ldots, r_k)$
- $I = \{(a_1, \ldots, a_n, c_{0,1}, \ldots, c_{0,k}) \mid a_1, \ldots a_n \in \{0, 1\}\}$
- $Act = \{\tau\}$
- $AP = \{x_1, \ldots, x_n, y_1, \ldots, y_m, r_1, \ldots, r_k\}$
- $L : S \to 2^{AP}$. where

$$\begin{aligned} L(a_1, \ldots, a_n, c_1, \ldots, c_n) \quad = \quad & \{x_1 \mid a_i = 1\} \cup \{r_j \mid c_j = 1\} \\ & \cup \{y_i \mid s \models \lambda_{y_i}(a_1, \ldots, a_n, c_1, \ldots, c_k) = 1\}, \end{aligned}$$

where $\lambda_{y_i} : S \to \{0, 1\}$ is the Boolean function obtained from the circuit for each y_i. The transitions are determined by the transition function of the registers δ_{r_j} that results from the circuit such that

$$(a_1, \ldots, a_n, c_1, \ldots, c_k) \xrightarrow{\tau} (a'_1, \ldots, a'_n, c'_1, \ldots, c'_k),$$

where $c'_j = \delta_{r_j}(a_1, \ldots, a_n, c_1, \ldots, c_k)$ and a'_is are nondeterministically decided.

Software Programs and Conditional Transitions

if $x \mod 2 = 1$ **then**
  $x \leftarrow x + 1$
else
  $x \leftarrow 2x$

- To model program fragments with data structures we need to consider conditional transitions.
- For that one needs to associate a program graph (an operational semantics) from which a transition system can be build

Program Graph

Let Var be a set of typed variables which values are given by an evaluation function $\eta \in Eval(Var)$, being $dom(x)$ the type of $x \in Var$. In a program graph:

- nodes are control components, i.e. a location ℓ is the point of the program that is being executed. In that point the evaluation function η simulates the state of the memory
- edges are labelled with conditions over the variables and by actions;
- A condition $g \in Cond(Var)$ where $Cond(Var)$ is the set of Boolean condictions over Var with atomic formulae

  $$\bar{x} \in \bar{D}$$

  being $\bar{x} = (x_1, \ldots, x_n)$ e $\bar{D} \subseteq dom(x_1) \times \cdots \times dom(x_n)$.

- Let $(-3 < x - x' \leq 5) \wedge (x \leq 2x') \wedge y = green$ then $dom(x) = dom(x') = \mathbb{Z}$ and e.g. $dom(y) = \{red, green\}$. The condition $(-3 < x - x' \leq 5$ corresponds to

  $$(x, x') \in \{(n, m) \in \mathbb{N}^2 \mid -3 < n - m \leq 5\}$$

  . And $y = green$ to $y \in \{green\}$.

- the *effect* of act action (command) is a function

  $$Effect : Act \times Eval(Var) \to Eval(Var)$$

  .

- If α is $x \leftarrow x + 1$ and $\eta(x) = 3$ then $Effect(\alpha, \eta)(x) = 4$.

Program Graph

A program graph PG over a set Var of typed variables is a tuple $(Loc, Act, Effect, \hookrightarrow, Loc_0, g_0)$

- Loc, set of locations (program points)
- Act, set of actions (commands)
- $Eval(Var)$ set of variable evaluations
- $Effect : Act \times Eval(Var) \to Eval(Var)$
- $\hookrightarrow \subseteq Loc \times Cond(Var) \times Act \times Loc$ condicional transition relation
- $Loc_0 \subseteq Loc$ set of initial locations

- $g_0 \in Cond(Var)$ initial condition

If $(\ell, g : \alpha, \ell') \in \hookrightarrow$ we write $\ell \overset{g:\alpha}{\hookrightarrow} \ell'$ and g is a *guard*, which can be omitted if it is a tautology. If the guard is not true the transition is not taken and the system blocks.

Vending Machine - V

Suppose that the machine counts the number of beverages and returns the money if it is empty.

- $Var = \{nsoda, nbeer\}$ with domain $\{0, \ldots, max\}$.
- Then $\eta \in Eval(Var)$ is any function that assigns $nsoda$ and $nbeer$ a value in that domain.
- $Loc = \{start, select\}$
- $Loc_0 = \{start\}$
- $Act = \{bget, sget, coin, ret\_coin, refill\}$
-

$$\begin{aligned} Effect(coin, \eta) &= Effect(ret\_coin, \eta) = \eta \\ Effect(sget, \eta) &= \eta[nsoda = nsoda - 1] \\ Effect(bget, \eta) &= \eta[nbeer = nbeer - 1] \\ Effect(refill, \eta) &= \eta[nsoda = max, nbeer = max] \end{aligned}$$

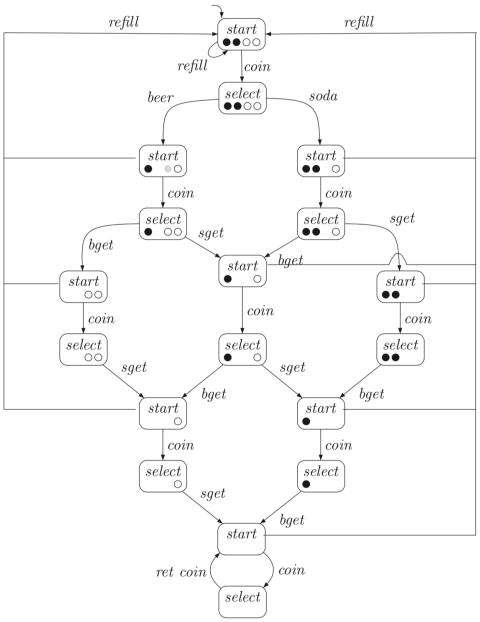
- g_0 é $(nsoda = max \wedge nbeer = max)$

Vending Machine with counting

- we have the following transitions $\hookrightarrow$:
- $start \overset{true:coin}{\hookrightarrow} select$
- $start \overset{true:refill}{\hookrightarrow} start$
- $select \overset{nsoda>0:sget}{\hookrightarrow} start$
- $select \overset{nbeer>0:bget}{\hookrightarrow} start$
- $select \overset{nsoda=0 \wedge nbeer=0:ret\_coin}{\hookrightarrow} start$

PG is not a transition system but can be unfolded to one given the values of the variables (e.g. $max = 2$). Suppose that $\bullet$ is one beer and $\circ$ one soda.

Vending Machine with Counting



Transition System of a Program Graph

- the states are pairs of control components, i.e. locations ℓ and a evaluation function η (it simulates the memory and the program counter). I.e $\langle \ell, \eta \rangle$
- the initial states are the states that satisfy the initial condition
- AP has the locations (to know where the program is) and the Boolean conditions over the variables. Ex $(x \leq 5) \wedge (\ell \in \{1, 2\}$
- The transitions are such that if $\ell \overset{g:\alpha}{\hookrightarrow} \ell'$ is a conditional transition in PG and $\eta \models g$ then there exists a transition by α from $\langle \ell, \eta \rangle$ to $\langle \ell', Effect(\alpha, \eta) \rangle$

Transition System of a Program Graph

Let $PG = (Loc, Act, Effect, \hookrightarrow, Loc_0, g_0)$, the transition system associated is $T(PG) = (S, Act, \longrightarrow, I, AP, L)$ where

- $S = Loc \times Eval(Var)$
- $\longrightarrow \subseteq S \times Act \times S$ defined by

$$\frac{\ell \overset{g:\alpha}{\hookrightarrow} \ell' \wedge \eta \models g}{\langle \ell, \eta \rangle \overset{\alpha}{\longrightarrow} \langle \ell', Effect(\alpha, \eta) \rangle}$$

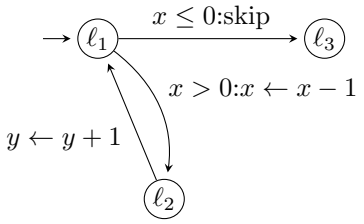
where $\eta \models g$ means that g is true for the evaluation η.

- $I = \{ \langle \ell, \eta \rangle \mid \ell \in Loc_0, \eta \models g_0 \}$
- $AP = Loc \cup Cond(Var)$
- $L(\langle \ell, \eta \rangle) = \{\ell\} \cup \{ g \in Cond(Var) \mid \eta \models g \}$

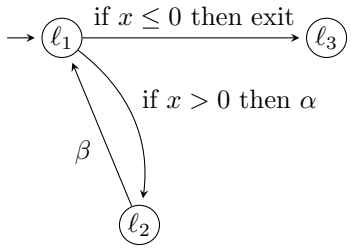
Example

while $x > 0$ **do**
$x \leftarrow x - 1$
$y \leftarrow y + 1$

Build a program graph identifying locations and actions.
The program graph can be:

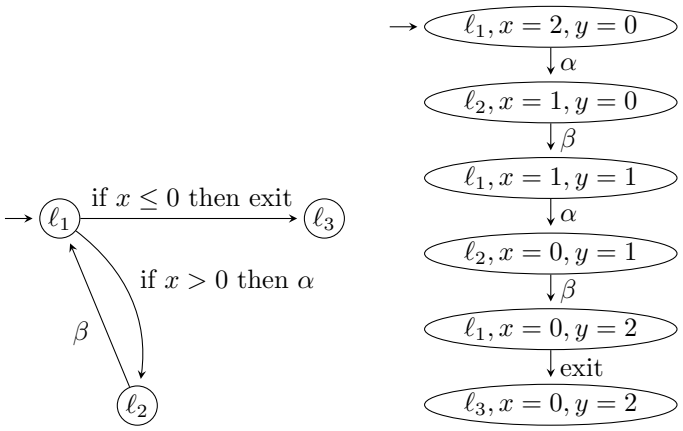


Let $\alpha = x \leftarrow x - 1$, $\beta = y \leftarrow y + 1$



Example

Determine the transition system if we start with $x = 2, y = 0$ and $\alpha = x \leftarrow x-1$, $\beta = y \leftarrow y + 1$.



Referências

[BA08] Mordechai Ben-Ari. *Principles of the Spin Model Checker*. Springer, 2008.

[BBF+01] B. Berard, M. Bidoit, A. Finkel, F. Laroussinie, A. Petit, L. Petrucci, and P. Schnoebelen. *Systems and Software Verification*. Springer-Verlag, 2001.

[BKL08] Christel Baier, Joost-Pieter Katoen, and Kim Guldstrand Larsen. *Principles of Model Checking*. MIT Press, 2008.

[HR04] Michael Huth and Mark Ryan. *Logic in Computer Science: Modelling and reasoning about systems*. CUP, 2004.

[Var94] M. Vardi. An automata-theoretic approach to linear temporal logic. In *Banff'94*, 1994.

[Var06] Moshe Vardi. Automata-theoretic techniques for temporal reasoning. In Patrick Blackburn, Johan van Benthem, and Frank Wolter, editors, *Handbook of Modal Logic*. Elsevier, 2006.

[VW07] M. Vardi and T. Wilke. Automata: From logics to algorithms. In *WAL 07*, pages 645–753, 2007.