Formal Specification and Verification

Formal specification (2)

26.11.2018

Viorica Sofronie-Stokkermans

e-mail: sofronie@uni-koblenz.de

Until now

- Logic
- Formal specification (generalities)

  Algebraic specification

Formal specification

- **Specification languages for describing programs/processes/systems**

  Model based specification

  transition systems, abstract state machines, specifications based on set theory

  Axiom-based specification

  algebraic specification

  Declarative specifications

  logic based languages (Prolog)

  functional languages, λ-calculus (Scheme, Haskell, OCaml, ...)

  rewriting systems (very close to algebraic specification): ELAN, SPIKE, ...

- **Specification languages for properties of programs/processes/systems**

  Temporal logic

Formal specification

- **Specification languages for describing programs/processes/systems**

  Model based specification

  transition systems, abstract state machines, specifications based on set theory

  Axiom-based specification

  algebraic specification

  Declarative specifications

  logic based languages (Prolog)

  functional languages, λ-calculus (Scheme, Haskell, OCaml)

  rewriting systems (very close to algebraic specification): ELAN, SPIKE

- **Specification languages for properties of programs/processes/systems**

  Temporal logic

Algebraic Specification

"A gentle introduction to CASL"

M. Bidoit and P. Mosses

`http://www.lsv.ens-cachan.fr/~bidoit/GENTLE.pdf`

Formal specification

- **Specification languages for describing programs/processes/systems**

  Model based specification

    transition systems, abstract state machines, specifications based on set theory

  Axiom-based specification

    algebraic specification

  Declarative specifications

    logic based languages (Prolog)

    functional languages, λ-calculus (Scheme, Haskell, OCaml)

    rewriting systems (very close to algebraic specification): ELAN, SPIKE

- **Specification languages for properties of programs/processes/systems**

  Temporal logic

Transition systems

Transition systems

- Executions
- Modeling data-dependent systems

Transition systems

- Model to describe the behaviour of systems
- Digraphs where nodes represent states, and edges model transitions
- **State:** Examples
  - the current colour of a traffic light
  - the current values of all program variables + the program counter
  - the current value of the registers together with the values of the input bits
- **Transition** ("state change"): Examples
  - a switch from one colour to another
  - the execution of a program statement
  - the change of the registers and output bits for a new input

Transition systems

Definition.

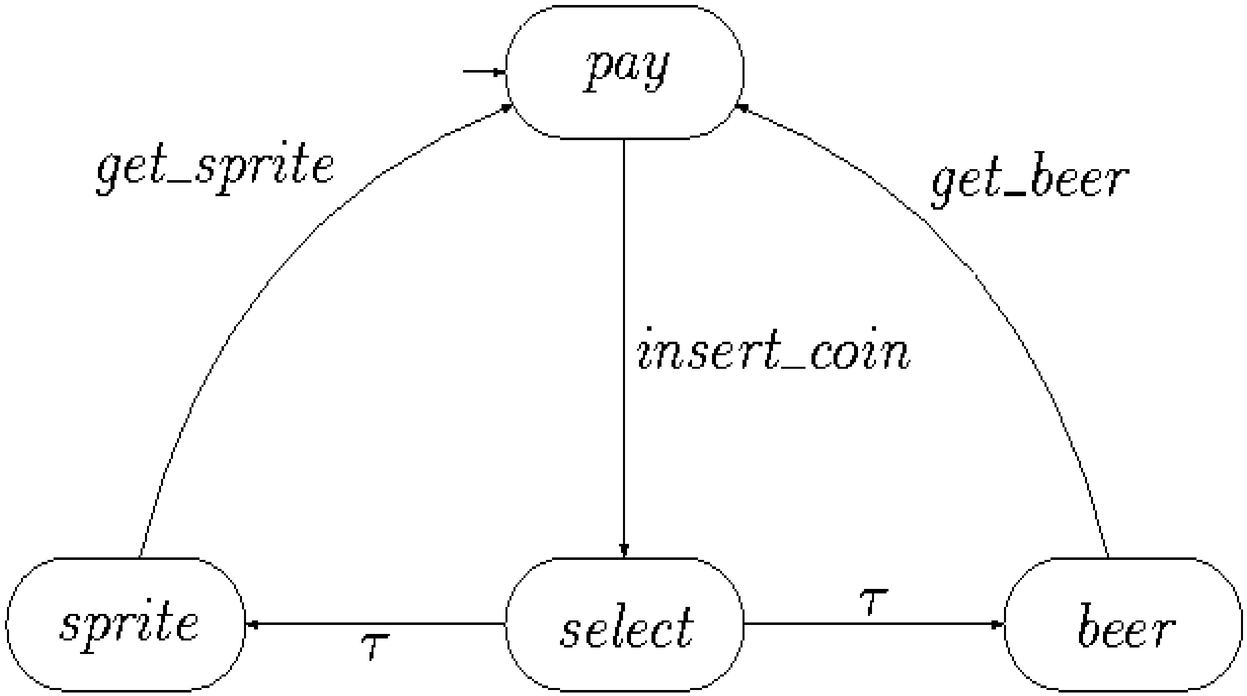
A transition system TS is a tuple $(S, Act, \rightarrow, I, AP, L)$ where:

- S is a set of states
- Act is a set of actions
- $\rightarrow \subseteq S \times Act \times S$ is a transition relation
- $I \subseteq S$ is a set of initial states
- AP is a set of atomic propositions
- $L : S \rightarrow 2^{AP}$ is a labeling function

S and Act are either finite or countably infinite

Notation: $s \stackrel{\alpha}{\rightarrow} s'$ instead of $(s, \alpha, s') \in \rightarrow$.

A beverage vending machine



states? actions?, transitions?, initial states?

Direct successors and predecessors

$Post(s, \alpha) = \{ s' \in S \mid s \overset{\alpha}{\to} s' \}$, $\quad Post(s) = \bigcup_{\alpha \in Act} Post(s, \alpha)$

$Pre(s, \alpha) = \{ s' \in S \mid s' \overset{\alpha}{\to} s \}$, $\quad Pre(s) = \bigcup_{\alpha \in Act} Pre(s, \alpha)$

$Post(C, \alpha) = \bigcup_{s \in C} Post(s, \alpha)$,

$Post(C) = \bigcup_{\alpha \in Act} Post(C, \alpha) \quad$ for $C \subseteq S$

$Pre(C, \alpha) = \bigcup_{s \in C} Pre(s, \alpha)$,

$Pre(C) = \bigcup_{\alpha \in Act} Pre(C, \alpha) \quad$ for $C \subseteq S$

State s is called terminal if and only if $Post(s) = \varnothing$

Action- and AP-determinism

Definition. Transition system $TS = (S, Act, \rightarrow, I, AP, L)$ is action-deterministic iff:

$$| \, I \, | \leq 1 \text{ and } | \, Post(s, \alpha) \, | \leq 1 \text{ for all } s \in S, \alpha \in Act$$

(at most one initial state and for every action, a state has at most one successor)

Definition. Transition system $TS = (S, Act, \rightarrow, I, AP, L)$ is AP-deterministic iff

- $| \, I \, | \leq 1$, and
- for all $s \in S, A \subseteq AP$: $\quad | \, Post(s) \cap \{ s' \in S \mid L(s') = A \} \, | \leq 1$

(at most one initial state; for every state s and every $A \subseteq AP$ there exists at most a successor of s in which all atomic propositions in A hold)

Non-determinism

Nondeterminism is a feature!

- to model concurrency by interleaving
  - no assumption about the relative speed of processes
- to model implementation freedom
  - only describes what a system should do, not how
- to model under-specified systems, or abstractions of real systems
  - use incomplete information

Non-determinism

Nondeterminism is a feature!

- to model concurrency by interleaving
  - no assumption about the relative speed of processes
- to model implementation freedom
  - only describes what a system should do, not how
- to model under-specified systems, or abstractions of real systems
  - use incomplete information

In automata theory, nondeterminism may be exponentially more succinct but that's not the issue here!

Transition systems $\neq$ finite automata

As opposed to finite automata, in a transition system:

- there are no accept states
- set of states and actions may be countably infinite
- may have infinite branching
- actions may be subject to synchronization
- nondeterminism has a different role

Transition systems are appropriate for modelling reactive system behaviour

Executions

- A finite execution fragment ρ of TS is an alternating sequence of states and actions ending with a state:

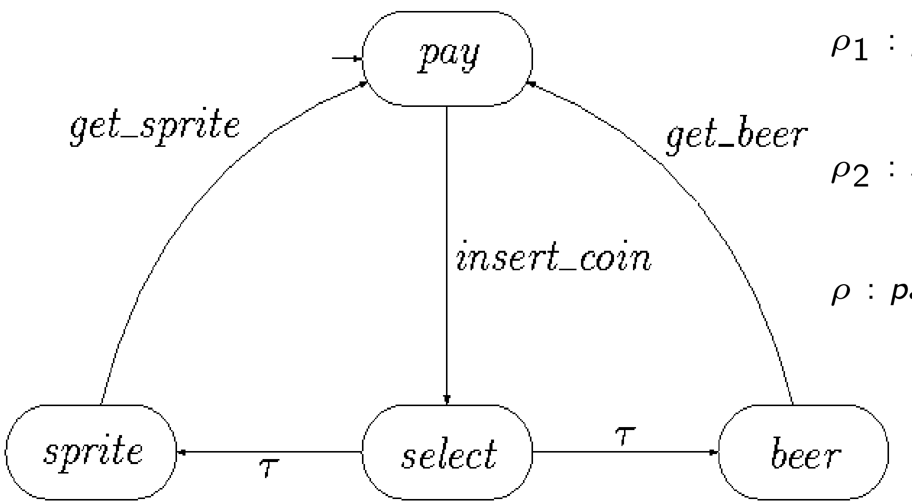
  $\rho = s_0 \alpha_1 s_1 \alpha_2 ... \alpha_n s_n$ such that $s_i \xrightarrow{\alpha_{i+1}} s_{i+1}$ for all $0 \leq i < n$.

- An infinite execution fragment ρ of TS is an infinite, alternating sequence of states and actions:

  $\rho = s_0 \alpha_1 s_1 \alpha_2 s_2 \alpha_3 ...$ such that $s_i \xrightarrow{\alpha_{i+1}} s_{i+1}$ for all $0 \leq i$.

- An execution of TS is an initial, maximal execution fragment
  - a maximal execution fragment is either finite ending in a terminal state, or infinite
  - an execution fragment is initial if $s_0 \in I$

Examples of Executions

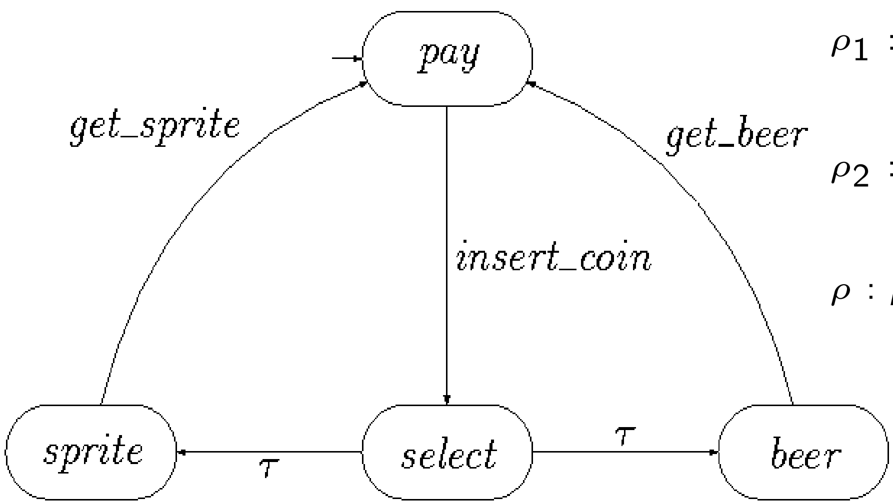


ρ_1 : pay $\xrightarrow{coin}$ select $\xrightarrow{\tau}$ sprite $\xrightarrow{sget}$ pay $\xrightarrow{coin}$ select $\xrightarrow{\tau}$ sprite $\xrightarrow{sget}$ . .

ρ_2 : select $\xrightarrow{\tau}$ sprite $\xrightarrow{sget}$ pay $\xrightarrow{coin}$ select $\xrightarrow{\tau}$ beer $\xrightarrow{bget}$ . . .

ρ : pay $\xrightarrow{coin}$ select $\xrightarrow{\tau}$ sprite $\xrightarrow{sget}$ pay $\xrightarrow{coin}$ select $\xrightarrow{\tau}$ sprite

Examples of Executions



$$\rho_1 : \textit{pay} \xrightarrow{\textit{coin}} \textit{select} \xrightarrow{\tau} \textit{sprite} \xrightarrow{\textit{sget}} \textit{pay} \xrightarrow{\textit{coin}} \textit{select} \xrightarrow{\tau} \textit{sprite} \xrightarrow{\textit{sget}} \ldots$$

$$\rho_2 : \textit{select} \xrightarrow{\tau} \textit{sprite} \xrightarrow{\textit{sget}} \textit{pay} \xrightarrow{\textit{coin}} \textit{select} \xrightarrow{\tau} \textit{beer} \xrightarrow{\textit{bget}} \ldots$$

$$\rho : \textit{pay} \xrightarrow{\textit{coin}} \textit{select} \xrightarrow{\tau} \textit{sprite} \xrightarrow{\textit{sget}} \textit{pay} \xrightarrow{\textit{coin}} \textit{select} \xrightarrow{\tau} \textit{sprite}$$

- Execution fragments ρ_1 and ρ are initial, but ρ_2 is not.
- ρ is not maximal as it does not end in a terminal state.
- Assuming that ρ_1 and ρ_2 are infinite, they are maximal

Reachable states

Definition. State $s \in S$ is called reachable in TS if there exists an initial, finite execution fragment

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

Reach(TS) denotes the set of all reachable states in TS.

Detailed description of states

Variables; Predicates

Beverage vending machine revisited

"Abstract" transitions:

$$start \xrightarrow{true:coin} select \quad \text{and} \quad start \xrightarrow{true:refill} start$$

$$select \xrightarrow{nsprite>0:sget} start \quad \text{and} \quad select \xrightarrow{nbeer>0:bget} start$$

$$select \xrightarrow{nsprite=0 \land nbeer=0:ret\text{-}coin} start$$

Action	Effect on variables
coin	
ret-coin	
sget	$nsprite := nsprite - 1$
bget	$nbeer := nbeer - 1$
refill	$nsprite := max;\ nbeer := max$

Program graph representation

Program graph representation

Some preliminaries

- typed variables with a valuation that assigns values in a fixed structure to variables
  - e.g., $\beta(x) = 17$ and $\beta(y) = -2$
- Boolean conditions: set of formulae over *Var*
  - propositional logic formulas whose propositions are of the form "$x \in D$"
  - $(-3 < x \leq 5) \wedge (y = green) \wedge (x \leq 2 * x')$
- effect of the actions is formalized by means of a mapping:

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

  - e.g., $\alpha \equiv x := y + 5$ and evaluation $\beta(x) = 17$ and $\beta(y) = -2$
  - $Effect(\alpha, \beta)(x) = \beta(y) + 5 = 3$,
  - $Effect(\alpha, \beta)(y) = \beta(y) = -2$

Program graph representation

Program graphs

A program graph PG over set Var of typed variables is a tuple

$$(Loc, Act, Effect, \rightarrow, Loc_0, g_0)$$

where

- Loc is a set of locations with initial locations $Loc_0 \subseteq Loc$
- Act is a set of actions
- $Effect : Act \times Eval(Var) \rightarrow Eval(Var)$ is the effect function
- $\rightarrow \; \subseteq \; Loc \times (\underbrace{Cond(Var)}_{\text{Boolean conditions on } Var} \times Act) \times Loc$, transition relation
- $g_0 \in Cond(Var)$ is the initial condition.

Notation: $l \stackrel{g:\alpha}{\rightarrow} l'$ denotes $(l, g, \alpha, l') \in \rightarrow$.

Beverage Vending Machine

- $Loc = \{start, select\}$ with $Loc_0 = \{start\}$
- $Act = \{bget, sget, coin, ret\text{-}coin, refill\}$
- $Var = \{nsprite, nbeer\}$ with domain $\{0, 1, ..., max\}$
- $Effect : Act \times Eval(Var) \rightarrow Eval(Var)$ defined as follows:

$$
\begin{aligned}
Effect(coin, \beta) &= \beta \\
Effect(ret\text{-}coin, \beta) &= \beta \\
Effect(sget, \beta) &= \beta[nsprite \mapsto \beta(nsprite) - 1] \\
Effect(bget, \beta) &= \beta[nbeer \mapsto \beta(nbeer) - 1] \\
Effect(refill, \beta) &= \beta[nsprite \mapsto max, nbeer \mapsto max]
\end{aligned}
$$

- $g_0 = (nsprite = max \wedge nbeer = max)$

From program graphs to transition systems

- Basic strategy: unfolding
  - state = location (current control) l + data valuation β $\qquad (l, \beta)$
  - initial state = initial location + data valuation satisfying the initial condition g_0
- Propositions and labeling
  - propositions: "at l" and "$x \in D$" for $D \subseteq dom(x)$
  - $< l, \beta >$ is labeled with "at l" and all conditions that hold in β.
- $l \overset{g:\alpha}{\rightarrow} l'$ and g holds in β then $< l, \beta > \overset{\alpha}{\rightarrow} < l', Effect(< l, \beta >) >$

Transition systems for program graphs

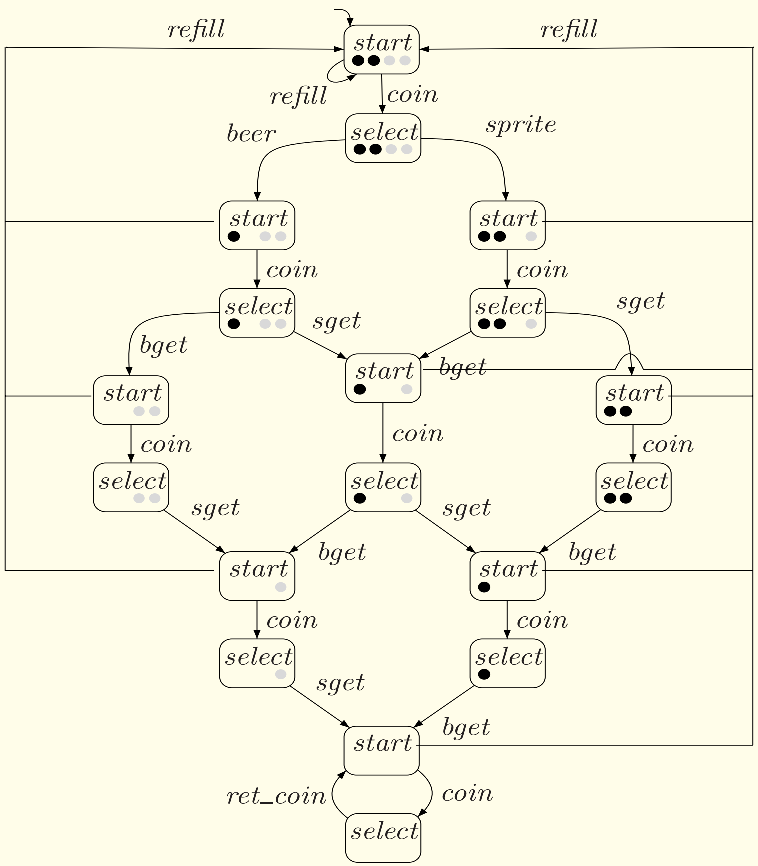
The transition system $TS(PG)$ of program graph

$$PG = (Loc, Act, Effect, \rightarrow, Loc_0, g_0)$$

over set Var of variables is the tuple $(S, Act, \rightarrow, I, AP, L)$ where:

- $S = Loc \times Eval(Var)$
- $\rightarrow S \times Act \times S$ is defined by the rule:
  If $l \stackrel{g:\alpha}{\rightarrow} l'$ and $\beta \models g$ then $< l, \beta > \stackrel{\alpha}{\rightarrow} < l', Effect(< l, \beta >) >$
- $I = \{< l, \beta >| \ l \in \mathrm{Loc}_0, \beta \models g_0\}$
- $AP = Loc \cup Cond(Var)$ and
- $L(< l, \beta >) = \{l\} \cup \{g \in Cond(Var) \mid \beta \models g\}$.

Transition systems for program graphs



Generalizations of transition systems

- More detailed description of states: Abstract state machines
- Emphasis on processes and their interdependency: CSP

- Durations: Timed automata
- Continuous evolution + discrete control: Hybrid automata

- Probabilistic systems: Markov chains, Probabilistic hybrid automata, ...