Formal Specification and Verification

Formal specification (2)

27.05.2014

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.

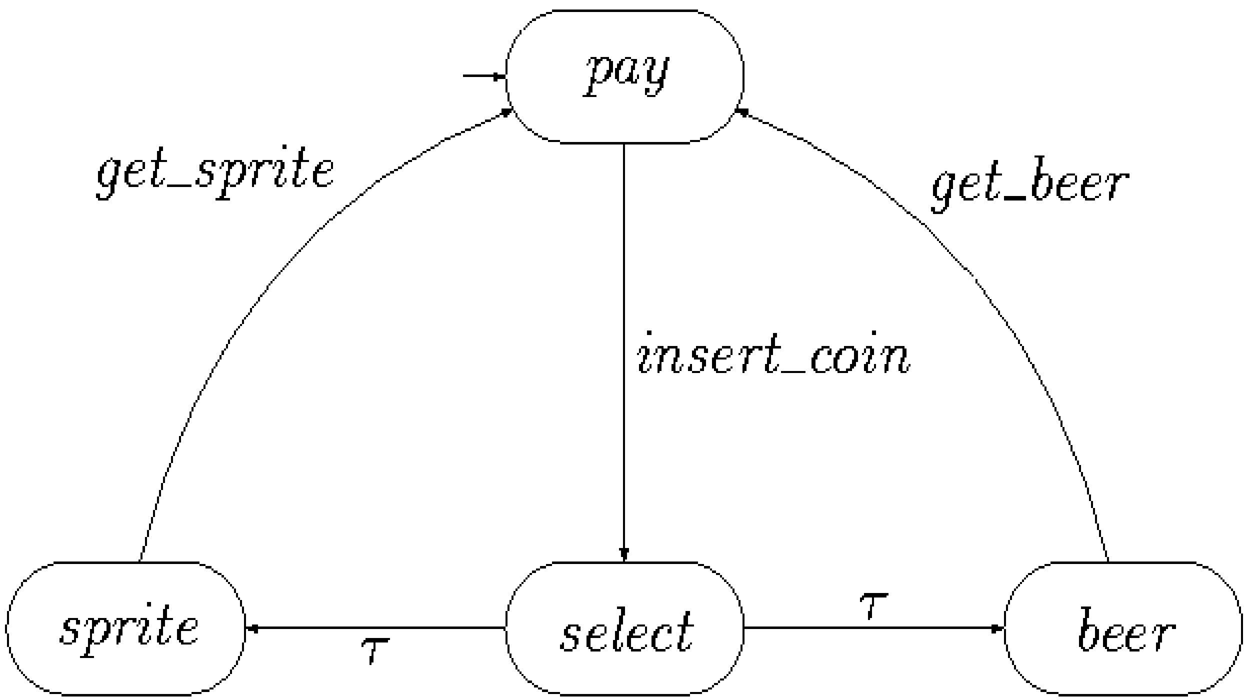
A transition system TS is a tuple $(S, Act, \rightarrow, I, AP, L)$ where:

- 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

$Post(s, \alpha) = \{s' \in S \mid s \xrightarrow{\alpha} s'\}$, $\qquad Post(s) = \bigcup_{\alpha \in Act} Post(s, \alpha)$

$Pre(s, \alpha) = \{s' \in S \mid s' \xrightarrow{\alpha} s\}$, $\qquad Pre(s) = \bigcup_{\alpha \in Act} Pre(s, \alpha)$

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

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

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

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

$$| \; I \; | \leq 1 \text{ and } | \; Post(s) \cap \{s' \in S \mid L(s') = A\} \; | \leq 1 \text{ for all } s \in S, A \in 2^{AP}$$

(at most one initial state; for state and every $A : AP \rightarrow \{0, 1\}$ there exists at most a successor of s in which "satisfies A")

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:

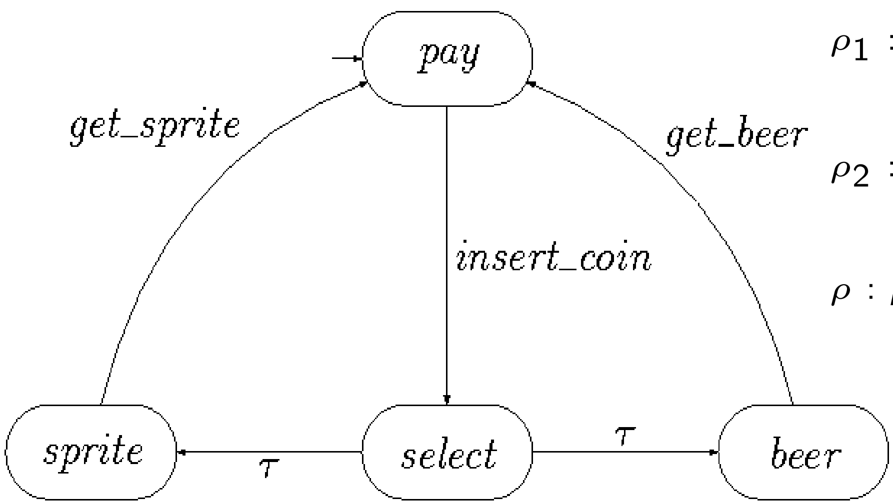
  $\rho = s_0 \alpha_1 s_1 \alpha_2 \ldots \alpha_n s_n$ such that $s_i \xrightarrow{\alpha_{i+1}} s_{i+1}$ for all $0 \leq i < n$.

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

  $\rho = s_0 \alpha_1 s_1 \alpha_2 s_2 \alpha_3 \ldots$ 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

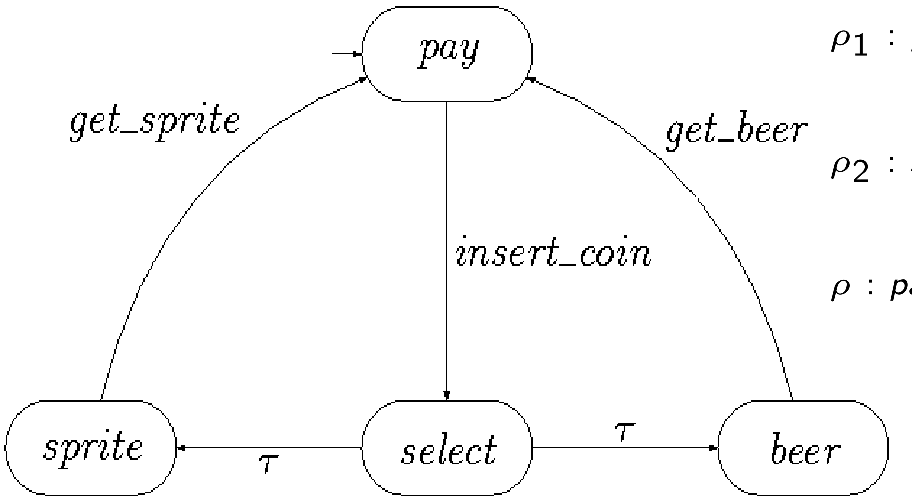


$$\rho_1 : \textit{pay} \xrightarrow{coin} \textit{select} \xrightarrow{\tau} \textit{sprite} \xrightarrow{sget} \textit{pay} \xrightarrow{coin} \textit{select} \xrightarrow{\tau} \textit{sprite} \xrightarrow{sget} \ldots$$

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

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

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

- 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:

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

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

$$\textit{select} \xrightarrow{\textit{nsprite}=0\wedge\textit{nbeer}=0:\textit{ret-coin}} \textit{start}$$

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

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:

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

  - 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: $l \overset{g:\alpha}{\rightarrow} l'$ 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 $\{0, 1, ..., max\}$
- $Effect : Act \times Eval(Var) \to Eval(Var)$ defined as follows:

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

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

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

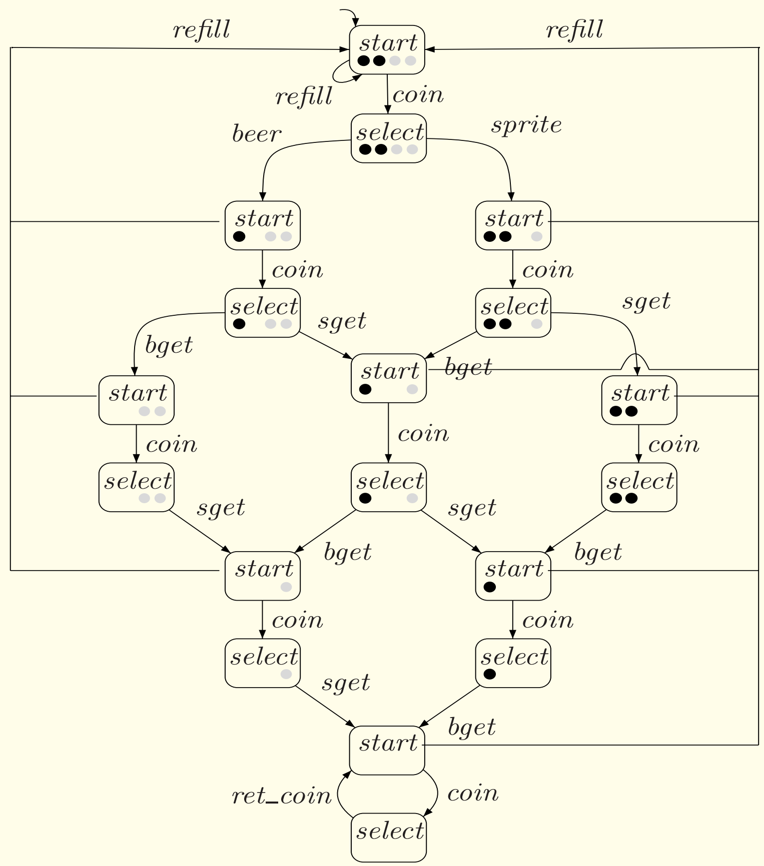
The transition system $TS(PG)$ of program graph

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

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 $l \overset{g:\alpha}{\rightarrow} l'$ and $\beta \models g$ then $< l, \beta > \overset{\alpha}{\rightarrow} < l', Effect(< l, \beta >) >$
- $I = \{< l, \beta >| \ l \in \mathrm{Loc}_0, \beta \models g_0\}$
- $AP = Loc \cup Cond(Var)$ 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

Abstract state machines (ASM)

Purpose

Formalism for modelling/formalising (sequential) algorithms

Not: Computability / complexity analysis

Invented/developed by

Yuri Gurevich, 1988

Old name

Evolving algebras

ASMs

Three Postulates

Sequential Time Postulate:

An algorithm can be described by defining a set of states, a subset of initial states, and a state transformation function

Abstract State Postulate:

States can be described as first-order structures

Bounded Exploration Postulate:

An algorithm explores only finitely many elements in a state to decide what the next state is. There is a finite number of names (terms) for all these “interesting” elements in all states.

Example: Computing Squares

Initial State

$square = 0$

$count = 0$

ASM for computing the square of input

if $input < 0$ then

    $input :=$ - $input$

else if $input > 0 \wedge$ $count < input$ then

  par

      $square := square + input$

      $count := count + 1$

  endpar

The Sequential Time Postulate

Sequential algorithm

An algorithm is associated with

- a set S of states
- a set $I \subseteq S$ of initial states
- A function $\tau : S \to S$
  (the one-step transformation of the algorithm)

Run (computation)

A run (computation) is a sequence $X_0, X_1, X_2 \ldots$ of states such that

- $X_0 \in I$
- $\tau(X_i) = X_{i+1}$ for all $i \geq 0$

Remark

Remark: In this formalism, algorithms are deterministic

$\tau : S \to S$ can be also viewed as a relation $R \subseteq S \times \{\tau\} \times S$ with

$$(s, \tau, s') \in R \text{ iff } \tau(s) = s'.$$

The Abstract State Postulate

States are first-order structures where

- all states have the same vocabulary (signature)
- the transformation τ does not change the base set (universe)
- S and I are closed under isomorphism
- if f is an isomorphism from a state X onto a state Y, then f is also an isomorphism from $\tau(X)$ onto $\tau(Y)$.

Example: Trees

Vocabulary

nodes:	unary, boolean:	the class of nodes (type/universe)
strings:	unary, boolean:	the class of strings
parent:	unary:	the parent node
firstChild:	unary:	the first child node
nextSibling:	unary:	the first sibling
label:	unary:	node label
c:	constant:	the current node

Vocabulary (Signature)

Signatures: A signature is a finite set of function symbols, where

- each symbol is assigned an arity $n \geq 0$
- symbols can be marked relational (predicates)
- symbols can be marked static (default: dynamic)

Vocabulary (Signature)

Signatures: A signature is a finite set of function symbols, where

- each symbol is assigned an arity $n \geq 0$
- symbols can be marked relational (predicates)
- symbols can be marked static (default: dynamic)

Remark: This is not a restriction

• predicates with arity n can be regarded as functions with arity

$s \dots s \rightarrow \text{bool}$

where s is the usual sort (for terms) and bool is a different sort

• The sort bool is described using a unary predicate Bool

• The sort Bool contains all formulae, in particular also $\top, \bot$ ("relational constants")

Vocabulary (Signature)

Signatures: A signature is a finite set of function symbols, where

- each symbol is assigned an arity $n \geq 0$
- symbols can be marked relational (predicates)
- symbols can be marked static (default: dynamic)

Each signature contains

- the constant *undef* ("undefined")
- the relational constants $\top$ (true), $\bot$ (false)
- the unary relational symbols $Boole, \neg$
- the binary relational symbols $\wedge, \vee, \rightarrow, \leftrightarrow, \approx$

These special symbols are all static

Vocabulary (Signature)

Signatures: A signature is a finite set of function/predicate symbols, where

- each symbol is assigned an arity $n \geq 0$
- symbols can be marked static (default: dynamic)

Each signature contains

- the constant *undef* ("undefined")
- the relational constants *true, false*
- the unary relational symbols *Boole*, $\neg$
- the binary relational symbols $\wedge, \vee, \rightarrow, \leftrightarrow, \approx$

These special symbols are all static

There is an infinite set of variables
Terms are built as usual from variables and function symbols
Formulae are built as usual

First-order Structures (States)

First-order structures (states) consist of

- a non-empty universe (called BaseSet)
- an interpretation of the symbols in the signature

Restrictions on states

- $0, 1, \mathit{undef} \in$ BaseSet (different)
- $\bot_{\mathcal{A}} = 0$, $\top_{\mathcal{A}} = 1$
- $\mathit{undef}_{\mathcal{A}} = \mathit{undef}$
- If f relational then $f_{\mathcal{A}} : \text{BaseSet} \to \{0, 1\}$
- $\mathit{Boole}_{\mathcal{A}} = \{0, 1\}$
- $\neg, \vee, \wedge, \to, \leftrightarrow$ are interpreted as usual

The reserve of a state

Reserve: Consists of the elements that are “unknown” in a state

The reserve of a state must be infinite

Extended States

Variable assignment

A function $\beta : \mathit{Var} \rightarrow \mathrm{BaseSet}$

(boolean variables are assigned 0 or 1)

Extended state

A pair $(\mathcal{A}, \beta)$ consisting of a state $\mathcal{A}$ and a variable assignment β.

Extended States

Variable assignment

A function $\beta : \mathit{Var} \rightarrow \mathrm{BaseSet}$

(boolean variables are assigned 0 or 1)

Extended state

A pair $(\mathcal{A}, \beta)$ consisting of a state $\mathcal{A}$ and a variable assignment β.

Evaluation of terms and formulae: as usual

Example: Trees

Vocabulary

nodes:	unary, boolean:	the class of nodes (type/universe)
strings:	unary, boolean:	the class of strings
parent:	unary:	the parent node
firstChild:	unary:	the first child node
nextSibling:	unary:	the first sibling
label:	unary:	node label
c:	constant:	the current node

Example: Trees

Terms

$parent(parent(c))$

$label(firstChild(c))$

$parent(firstChild(c)) = c$ (Boolean, formula)

$nodes(x) \to parent(x) = parent(nextSibling(x))$

(x is a variable)

Isomorphism

Lemma (Isomorphism)

Isomorphic states (structures) are indistinguishable by ground terms:

Justification for postulate

Algorithm must have the same behaviour for indistinguishable states

Isomorphic states are different representations of the same abstract state!

State updates

Locations. A location is a pair $(f, \overline{a})$ with

- f an n-ary function symbol
- $\overline{a} \in \mathrm{BaseSet}^n$ an n-tuple

Examples

(parent, a), $(\mathit{firstChild}, a)$, $(\mathit{nextSibling}, a)$, $(c,)$

State updates

Locations. A location is a pair $(f, \overline{a})$ with

- f an n-ary function symbol
- $\overline{a} \in \mathrm{BaseSet}^n$ an n-tuple

Examples

(parent, a), $(\mathit{firstChild}, a)$, $(\mathit{nextSibling}, a)$, $(c,)$

An update is a triple $(f, \overline{a}, b)$ with

- $(f, \overline{a})$ a location
- f not static
- $b \in \mathrm{BaseSet}$
- if f is relational, then $b \in \{0, 1\}$

State updates

Locations. A location is a pair $(f, \overline{a})$ with

- f an n-ary function symbol
- $\overline{a} \in \text{BaseSet}^n$ an n-tuple

Examples

$(parent, a)$, $(firstChild, a)$, $(nextSibling, a)$, $(c,)$

An update is a triple $(f, \overline{a}, b)$ with

- $(f, \overline{a})$ a location
- f not static
- $b \in \text{BaseSet}$
- if f is relational, then $b \in \{0, 1\}$

Intended meaning:

f is changed by changing $f(\overline{a})$ to b.

State updates

Locations. A location is a pair $(f, \overline{a})$ with

- f an n-ary function symbol
- $\overline{a} \in \text{BaseSet}^n$ an n-tuple

Examples

$(parent, a)$, $(firstChild, a)$, $(nextSibling, a)$, $(c,)$

An update is a triple $(f, \overline{a}, b)$ with

- $(f, \overline{a})$ a location
- f not static
- $b \in \text{BaseSet}$
- if f is relational, then $b \in \{tt, ff\}$

Intended meaning:

f is changed by changing $f(\overline{a})$ to b.

An update is trivial if $f_{\mathcal{A}}(\overline{a}) = b$

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

Timed automata

- transition systems + timing constraints

Timed automata

A timed automaton is a finite automaton extended with a finite set of real-valued clocks. During a run of a timed automaton, clock values increase all with the same speed. Along the transitions of the automaton, clock values can be compared to integers. These comparisons form guards that may enable or disable transitions and by doing so constrain the possible behaviors of the automaton. Further, clocks can be reset.

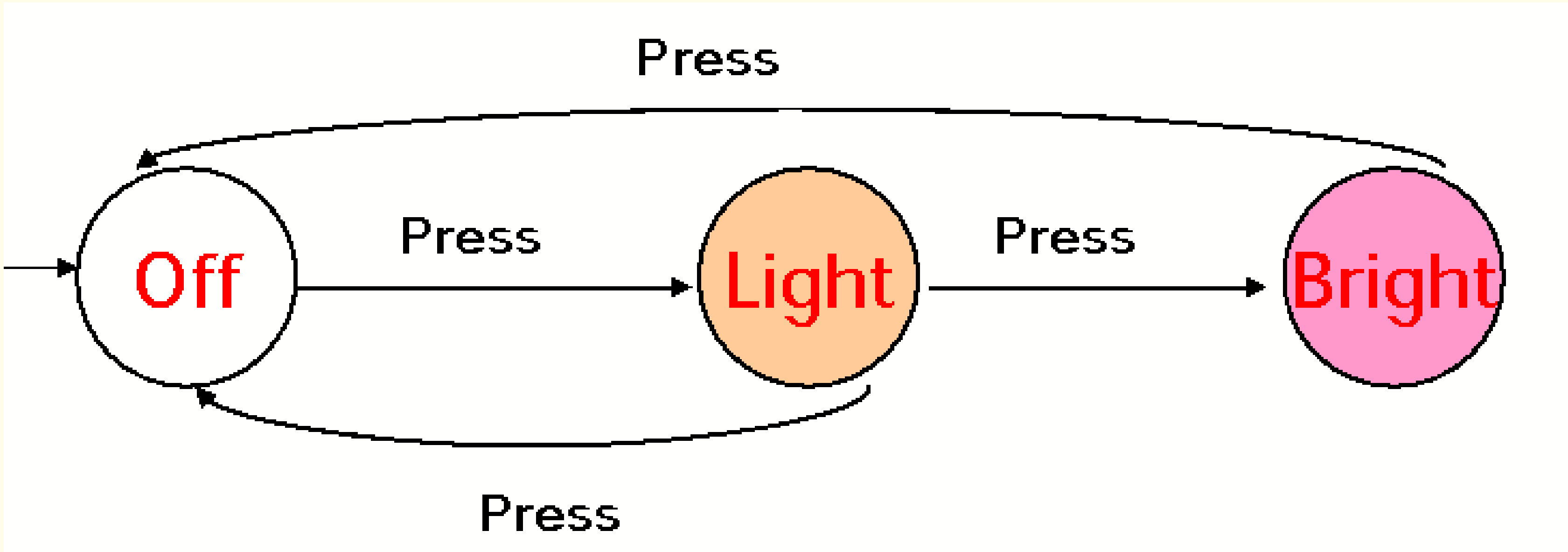
Timed automata

A timed automaton is a finite automaton extended with a finite set of real-valued clocks. During a run of a timed automaton, clock values increase all with the same speed. Along the transitions of the automaton, clock values can be compared to integers. These comparisons form guards that may enable or disable transitions and by doing so constrain the possible behaviors of the automaton. Further, clocks can be reset.

Timed automata can be used to model and analyse the timing behavior of computer systems, e.g., real-time systems or networks.

Timed automata

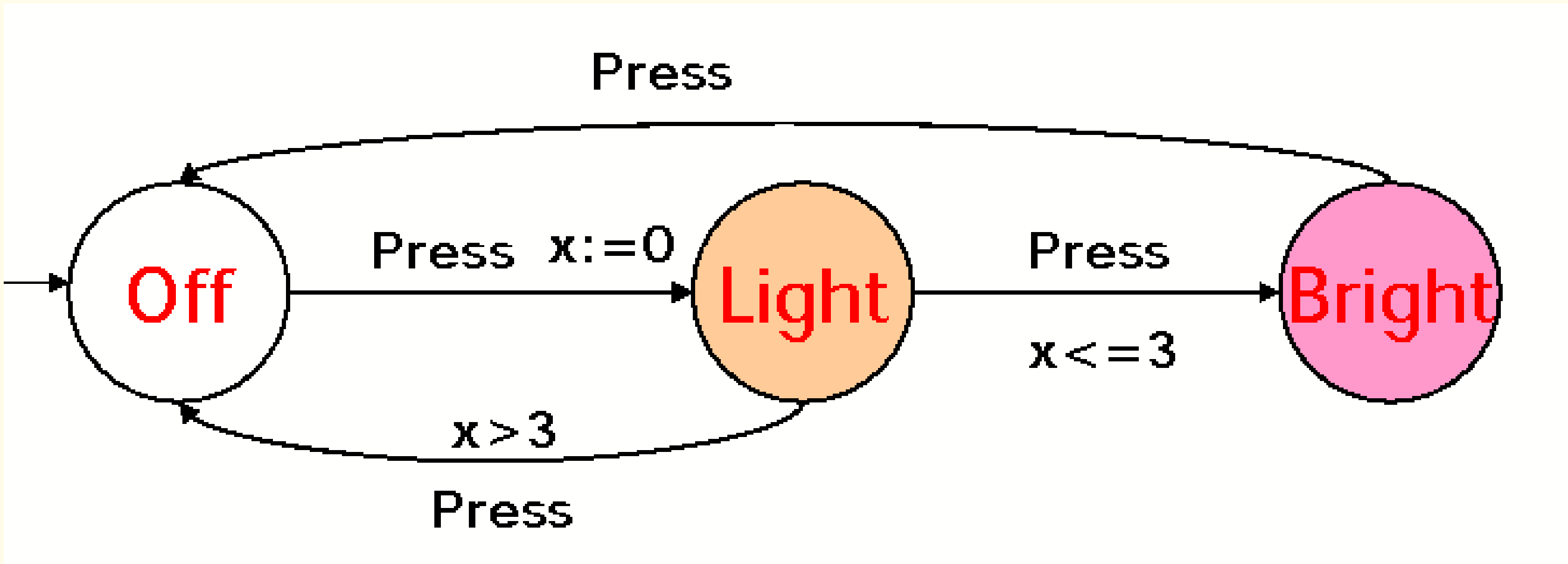
Example: Simple Light Control



WANT: if press is issued twice quickly then the light will get brighter; otherwise the light is turned off.

Timed automata

Example: Simple Light Control



Solution: Add a real-valued clock x

Adding continuous variables to transition systems

Timed automata: Syntax

- A finite set Loc of locations
- A subset $Loc_0 \subseteq Loc$ of initial locations
- A finite set Act of labels (alphabet, actions)
- A finite set X of clocks
- Invariant $Inv(l)$ for each location $l \in Loc$: (clock constraint over X)
- A finite set E of edges. Each edge has:
  - source location l, target location l'
  - label $a \in Act$ (empty labels also allowed)
  - guard g (a clock constraint over X)
  - a subset X' of clocks to be reset

Timed automata: Semantics

For a timed automaton

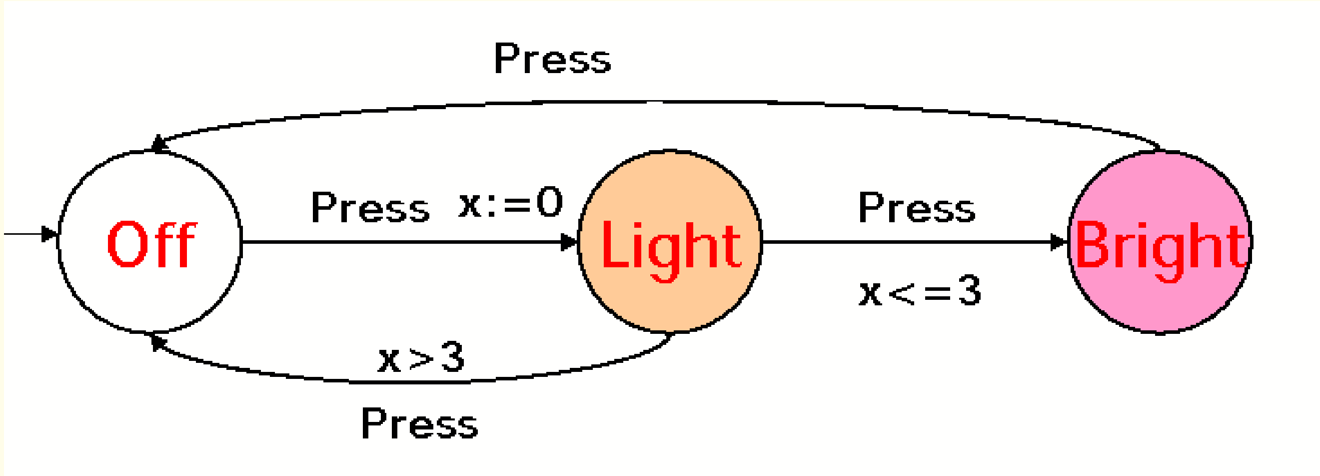
$$A = (Loc, Loc_0, Act, X, \{Inv_l\}_{l \in Loc}, E)$$

define an infinite state transition system $S(A)$:

- **States** S: a state s is a pair (l, v), where
  l is a location, and
  v is a clock vector, mapping clocks in X to $\mathbb{R}$, satisfying $Inv(l)$
- **Initial States:** (l, v) is initial state if l is in Loc_0 and $v(x) = 0$
- **Elapse of time transitions:** for each nonnegative real number d, $(l, v) \xrightarrow{d} (l, v + d)$ if both v and $v + d$ satisfy $Inv(l)$
- **Location switch transitions:** $(l, v) \xrightarrow{a} (l', v')$ if there is an edge (l, a, g, X', l') such that v satisfies g and $v' = v[\{x \mapsto 0 \mid x \in X'\}]$.

Timed automata

Example: Simple Light Control



Timed automaton:

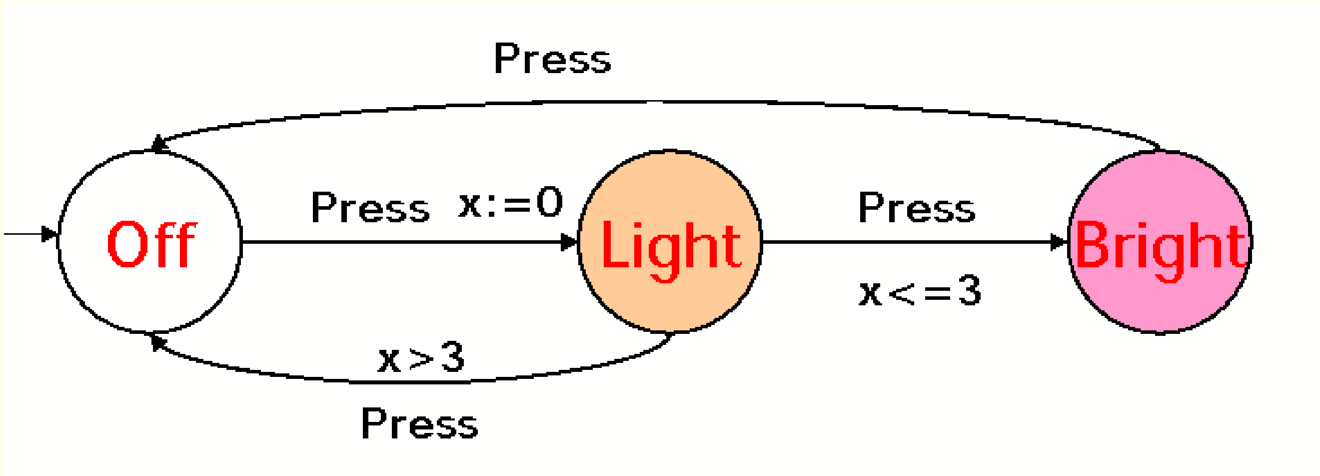
$Loc = \{\text{Off}, \text{Light}, \text{Bright}\}$, $Loc_0 = \{\text{Off}\}$, $\quad Act = \{\text{Press}\}$

$X = \{x\}$; $\text{Inv}(\text{Off}) = \text{Inv}(\text{Light}) = \text{Inv}(\text{Bright}) = (x \geq 0)$

Edges: $(\text{Off}, \text{Press}, \top, \{x\}, \text{Light})$, $(\text{Light}, \text{Press}, x > 3, \varnothing, \text{Off})$
$(\text{Light}, \text{Press}, x \leq 3, \varnothing, \text{Bright})$, $(\text{Bright}, \text{Press}, \top, \varnothing, \text{Off})$

Timed automata

Example: Simple Light Control



States: (Off, v), (Light, v), (Bright, v) (v value of clock x).
Initial state: $(\mathsf{Off}, 0)$.

Transitions (Examples)

Elapse of time: $(\mathsf{Off}, 10) \xrightarrow{5} (\mathsf{Off}, 15)$

Location switch: $(\mathsf{Off}, 10) \xrightarrow{\mathsf{Press}} (\mathsf{Light}, 0)$