Software Verification and Testing

Lecture Notes: Transition Systems I

Set-Based and Relational Formal Methods

context: sets and relations are powerful tools for analysing software systems

- specification: abstract and concise modelling of system properties
- verification: elegant calculational equational proofs in relational calculus
- mechanisation: automated or interactive machine proofs in FOL
- visualisation: properties of finite systems depictable by graphs
- push-bottom approach: powerful decision procedures for finite systems

objections:

- this is theory, but does it work in practice?
- who can handle the 573 rules of the relational calculus?
- is exhaustive search feasible for large finite systems?
- how to model data structures, data types, objects?

Set-Based and Relational Formal Methods

plan:

- we further consider relational structures for modelling properties of sequential systems
- we learn how to visualise finite systems by graphs
- we extend these methods to concurrent systems
- we formalise simple system properties in PL and FOL

Transition Systems

context: we have given a relational semantics to simple while-programs (aka regular programs)

$$
\begin{aligned}
\mathsf{skip} &= 1_A \\
\mathsf{abort} &= \emptyset \\
R; S &= R \circ S \\
R + S &= R \cup S \\
\mathsf{if}\ B\ \mathsf{then}\ R\ \mathsf{else}\ S &= B \lhd R \cup (A - B) \lhd S \\
\mathsf{while}\ B\ \mathsf{do}\ R &= (B \lhd R)^* \circ (A - B)
\end{aligned}
$$

Transition Systems

actions and propositions:

- sets model static aspects of a program
  - examples: tests and properties of the store (values of variables. . .)
  - sets can be identified with propositions, i.e., logical sentences
- relations model dynamic aspects of a program
  - examples: changes of the store (assignments to variables. . .)
  - relations can be identified with actions

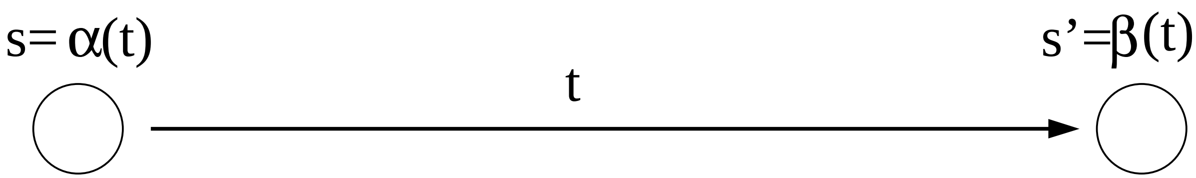
conclusion: the relational semantics of while-programs models transitions on a state space

question: how to model transition systems formally?

Transition Systems

definition: a transition system is a structure $\mathcal{A} = (S, T, \alpha, \beta)$ where

- S is a (finite or infinite) set of states
- T is a (finite or infinite) set of transitions
- α and β are functions from T to S such that, for every $t \in T$,
  - $\alpha(t)$ is the source of t
  - $\beta(t)$ is the target of t



Transition Systems

remarks:

- every pair $(\alpha(t), \beta(t))$ is a relation on S
- different transitions can have the same source and target

definition:

- a finite path of length n in a transition system $\mathcal{A}$ is a sequence $t_0, \ldots, t_{n-1}$ of transitions where $\beta(t_i) = \alpha(t_{i+1})$ holds $\forall i. 0 \leq i < n-1$
- an infinite path in $\mathcal{A}$ is a sequence $t_0, t_1 \ldots$ of transitions where $\beta(t_i) = \alpha(t_{i+1})$ holds $\forall i. i \geq 0$

intuition: paths are transitions glued together

Transition Systems

notation: we extend α and β to paths by setting

$$\alpha(t_0, \ldots, t_{n-1}) = \alpha(t_0) \qquad \beta(t_0, \ldots, t_{n-1}) = \beta(t_{n-1}) \qquad \alpha(t_0, t_1, \ldots) = \alpha(t_0)$$

path product: let $c = t_0, \ldots, t_{n-1}$ and $c' = t'_0, \ldots, t'_{m-1}$ be finite paths

- the product of c and c' is defined whenever $\beta(c) = \alpha(c')$ as

$$c \cdot c' = c = t_0, \ldots, t_{n-1}, t'_0, \ldots, t'_{m-1}$$

- $\alpha(c \cdot c') = \alpha(c)$ and $\beta(c \cdot c') = \beta(c')$
- for each state s we define the empty path ϵ_s by $\alpha(\epsilon_s) = \beta(\epsilon_s) = s$

remark: c' could as well be an infinite path, c couldn't

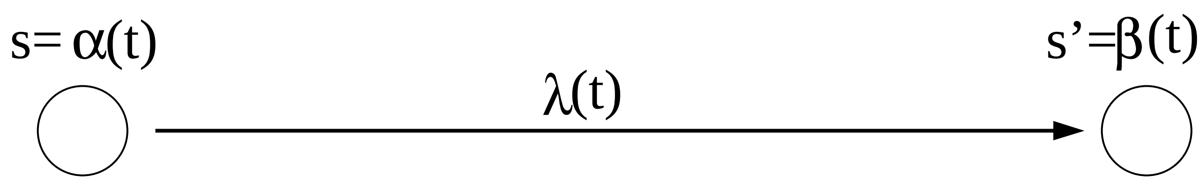
Labelled Transition Systems

problem: with transition systems we cannot model actions

definition: a labelled transition system (LTS) over an alphabet A is a structure $\mathcal{A} = (S, T, \alpha, \beta, \lambda)$ where

- (S, T, α, β) is a transition system and
- λ is a labelling function from T to A

intuition: $\lambda(t)$ indicates the action that triggers transition t



Labelled Transition Systems

assumption: we do not distinguish transitions with the same source, target and label

notation: we write $t : s \rightarrow_a s'$ to denote that transition t triggered by action a goes from state s into state s'

definition: let $c = t_0, t_1, \ldots$ be a path in an LTS $\mathcal{A}$. The trace of c is the sequence of actions

$$trace(c) = \lambda(t_0), \lambda(t_1), \ldots$$

Labelled Transition Systems

definition: let $c = t_0, t_1, \ldots$ be a path in an LTS $\mathcal{A}$. The run correcponding to c is the sequence of states

$$run(c) = \alpha(t_0), \alpha(t_1), \ldots$$

intuition: runs are sequences of states related by transitions

distinction:

- paths are sequences of transitions
- traces are sequences of actions
- runs are sequences of states

Traces and Regular Expressions

task: find a more compact notation for sets of traces

definition: regular expressions over alphabet A

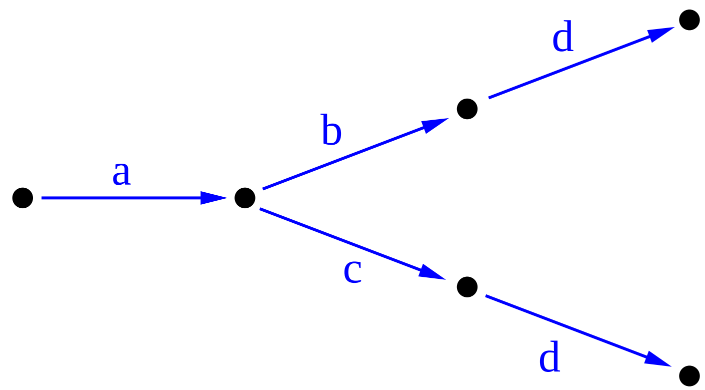
- if a is a letter from A, then a is a regular expression denoting the set $\{a\}$
- if e_1 and e_2 are regular expressions denoting sets E_1 and E_2 then
  - $e_1 \cdot e_2$ is a regular expression denoting $\{u_1 u_2 : u_i \in E_i\}$
  - $e_1 + e_2$ is a regular expression denoting $E_1 \cup E_2$
- if e is a regular expression denoting E then
  - e^* is a regular expression denoting $\epsilon \cup \{u_0 \dots u_{n-1} : u_i \in E \wedge n \geq 0\}$
  - e^ω is a regular expression denoting $\{u_0 u_1 \cdots : u_i \in E \wedge u_i \neq \epsilon\}$

notation: the operations $\cdot$, $+$ and * are called regular operations

remark: regular programs are programs built from the regular operations

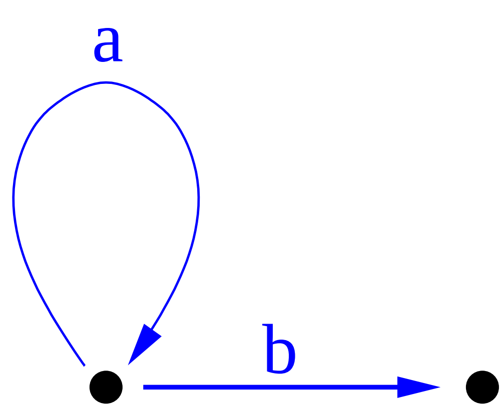
Traces and Regular Expressions

examples: the regular expression $a(b+c)d$ corresponds to the LTS



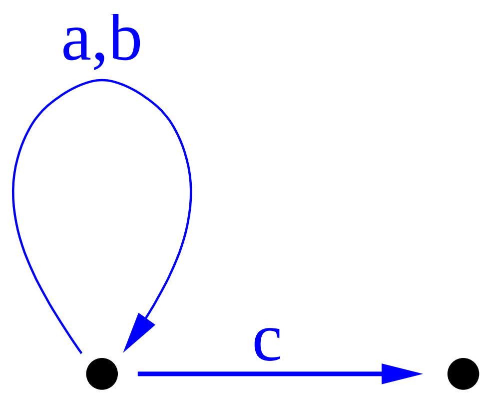
Traces and Regular Expressions

examples: the regular expression a^*b corresponds to the LTS



Traces and Regular Expressions

examples: the regular expression $(a + b)^*c$ corresponds to the LTS



Adding Labels for States

extension: instead of a LTS with labelling function λ for transitions we can also define LTS with labelling functions λ_σ and λ_τ for states and transitions

intuition: we can use state labels for explicitly identifying states with the set of atomic propositions that hold in that state

Examples

a boolean variable b

- states are labelled by $true$ and $false$
- values of the variables can be changed by assignment

$$t_1 : true \rightarrow_{b:=true} true \qquad t_2 : true \rightarrow_{b:=false} false$$

$$t_3 : false \rightarrow_{b:=true} true \qquad t_4 : false \rightarrow_{b:=false} false$$

- values of b can be tested

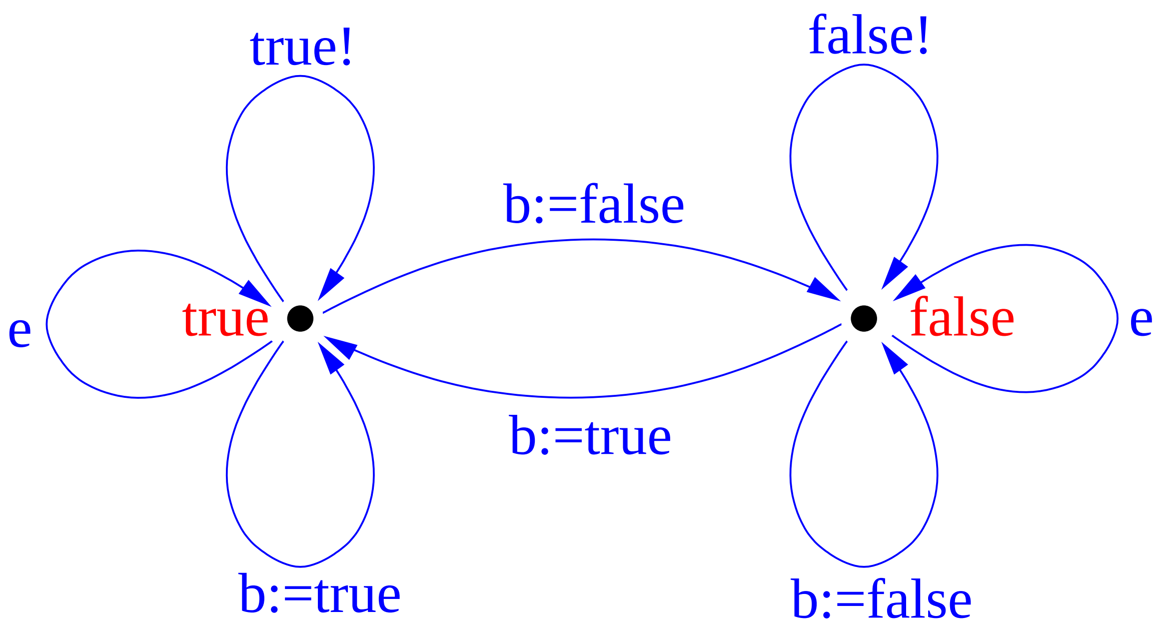
$$t_5 : true \rightarrow_{true!} true \qquad t_6 : false \rightarrow_{false!} false$$

- empty action e (or skip) can be added

$$t_7 : true \rightarrow_{\mathsf{skip}} true \qquad t_8 : false \rightarrow_{\mathsf{skip}} false$$

Examples

a boolean variable b



question: what if $true!$ is applied to $false$?

Examples

a counter with values 0, 1, 2, 3

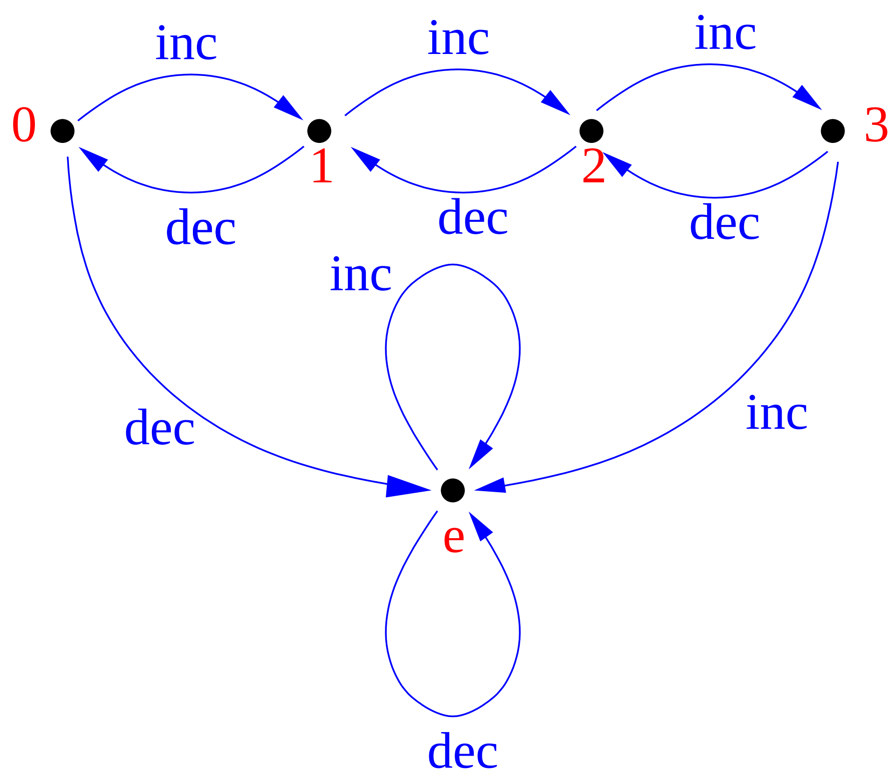
- obvious transitions (forgetting the t_i)

$$0 \longrightarrow_{inc} 1 \qquad 1 \longrightarrow_{inc} 2 \qquad 2 \longrightarrow_{inc} 3 \qquad 3 \longrightarrow_{dec} 2 \qquad 2 \longrightarrow_{dec} 1 \qquad 1 \longrightarrow_{dec} 0$$

- design decision:
  - disallow incrementing 3 and decrementing 0: no further transitions
  - counter modulo 4: add transitions $3 \longrightarrow_{inc} 0$ and $0 \longrightarrow_{dec} 3$
  - add error state e and transitions $3 \longrightarrow_{inc} e$, $0 \longrightarrow_{dec} e$, $e \longrightarrow_{inc} e$, $e \longrightarrow_{dec} e$
- actions like tests and $skip$ can also be added
- a set of initial states can be defined as $init = \{0\}$

Examples

a counter with values 0, 1, 2, 3 and error state



Examples

a bounded buffer with two slots used as a queue

- alphabet $\{a, b\}$
- states (labelled by possible contents): $empty$, a, b, aa, ab, ba, bb
- actions
  - enter letter in buffer if it is not full
  - remove letter from buffer if it is not empty

$$empty \rightarrow_{enq(a)} a \qquad empty \rightarrow_{enq(b)} b$$

$$a \rightarrow_{enq(a)} aa \qquad a \rightarrow_{enq(b)} ba \qquad b \rightarrow_{enq(a)} ab \qquad b \rightarrow_{enq(b)} bb$$

$$a \rightarrow_{deq} empty \qquad b \rightarrow_{deq} empty$$

$$aa \rightarrow_{deq} a \qquad ab \rightarrow_{deq} a \qquad ba \rightarrow_{deq} b \qquad bb \rightarrow_{deq} b$$

- etc.

Examples

a bounded buffer can you draw a diagram?

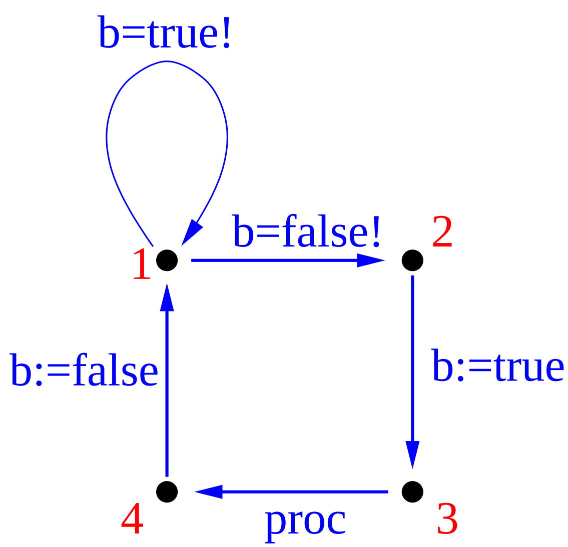
Examples

a sequential program: consider the pseudo-code fragment

```
while true do
  1: if not b then
     begin
       2: b:=true;
       3: proc;
       4: b:= false;
     end
```

Examples

a sequential program: use program counters as state labels



Examples

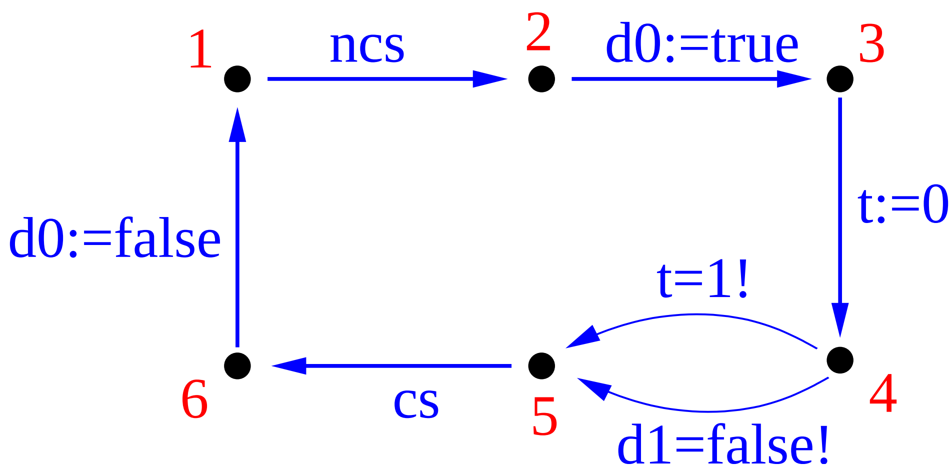
Peterson's mutex algorithm:

```
while true do
  begin
   1: non-critical section;
   2: d0:=true;
   3: turn:=0;
   4: wait(d1=false or turn=1);
   5: critical section;
   6: d0:=false;
 end
```

```
while true do
  begin
   1: non-critical section
   2: d1:=true;
   3: turn:=1;
   4: wait(d0=false or turn=0);
   5: critical section;
   6: d1:=false;
  end
```

Examples

Peterson's mutex algorithm: diagram for first process



LTSs and Relations

fact: different transitions cannot have the same source, target and action label

idea: fix the label, consider corresponding pairs of sources and targets

theorem: with each LTS $\mathcal{A} = (S, T, \alpha, \beta, \lambda)$ we can associate a relational structure $(S, \{R_{\lambda(t)} : t \in T\})$, where $R_{\lambda(t)} = \{(\alpha(t'), \beta(t')) \in \lambda(t) : t' \in T\}$

LTSs and Relations

remarks:

- conversely, relational structures can be turned into LTSs by assigning different transitions to all elements of the transition relations
- we often do not distinguish between LTSs and relational structures

definitions:

- we call R_a the transition relation associated with the action a
- a LTS is deterministic if all transition relations are partial functions

Trees

idea: trees are special relational structures

definition: a tree is a relational structure (S, R) where

- the set of nodes S contains a distinguished element r, the root of the tree and $(r, s) \in R^*$ holds for all $s \in S$
- for every $s \neq r$ there is a unique $s' \in S$ such that $(s', s) \in R$
- R is acyclic, that is for all $(t, t) \notin R^+$ for all $s \in S$

Trees

example: unwinding a finite LTS with initial state

- take runs of the LTS as nodes of the tree
- take the direct-prefix-relation on runs as the successor relation in the tree