Conception et vérification formelles de systèmes informatiques 1
-
Formal Design and Verification of Computer Systems 1
-
(INFO-F-412)

Thierry Massart
Université Libre de Bruxelles
Département d'Informatique

August 2009

Acknowledgment

I want to thank Edmund Clarke, Keijo Heljanko, Tommi Junttila, Jean-François Raskin, Klaus Schneider, and Jan Tretmans who allows me to access to their course's slides to prepare these notes

Thierry Massart

Chapters

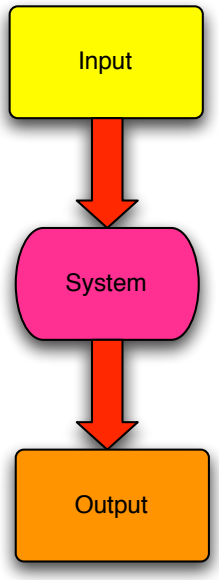
Main References

- Klaus Schneider ; Verification of Reactive Systems : Formal Methods and Algorithms, Springer Verlag, 2004.
- Béatrice Bérard, Michel Bidoit, Alain Finkel , François Laroussinie , Antoine Petit , Laure Petrucci and Philippe Schnoebelen ; Systems and Software Verification. Model-Checking Techniques and Tools, Springer, 2001.
- E.M. Clarke, O. Grumberg and D. Peled ; Model Checking, The MIT Press, 1999.

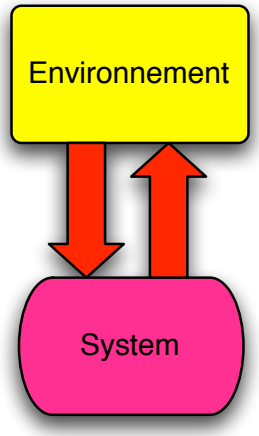
Chapter 1 : Introduction

Plan

Transformational vs Reactive Systems



Transformational system : computes results.



Interactive system : requests the environment for information
Reactive system : reacts to events from the environment.

Features of systems

Transformational or computational system

- specified by input-output relations
- can be specified e.g. with pre and post conditions in Floyd-Hoare tradition

Reactive system

- not-necessarily-terminating system (generally termination, called deadlock, is a bad system's state),
- must normally always be ready for interaction,
- the interaction is the basic unit of computation,
- it is generally specified by the triple : event - condition - action
- the sequence of interactions provides the computation
- the possible temporal ordering of actions determine the correctness,
- for real-time systems ; quantitative real-time aspects must also be taken into account.

Embedded systems

Found in

- safety-critical applications : automotive devices and controls, railways, aircraft, aerospace and medical devices
- 'mobile worlds' and 'e-worlds', the 'smart' home, factories ...

Features

- direct interaction with the environment
- environment : mechanical, electronic, ...
- through sensors/actuators
- multithreaded software
- often application-specific hardware/processors
- reconfigurable systems are emerging

Embedded systems

- uses more than 98% of the microprocessors shipped today
- causes up to 40% of development costs of modern cars
- enormous markets in consumer electronics, automotive & avionics industries
- growing field of applications
- growing impact on competition
- more "intelligent" systems
- 90% of new development in automotive is software

Example : Automotive Industry

- up to 100 embedded systems in modern cars
- connected with busses like CAN, TT-CAN, FlexRay, MOST
- Audi A8 has 90MB memory, the former model only had 3 MB
- Different applications : motor optimization (fuel injection), central locking unit, ABS (Antilock Brake System), EBD (Electronic Brake Distribution), EPS (Electronic Power Steering), ESP (Electronic Stability Program), parktronic,...

Strenghts and weaknesses of embedded systems

Strenghts of embedded systems

- Allows more flexible and intelligent devices

Weaknessess of embedded systems

- Discrete sytems $\Rightarrow$ very sensitive,
- Complex $\Rightarrow$ difficult to design,
- Embedded $\Rightarrow$ difficult to monitor,
- Safety critical $\Rightarrow$ must be correct.

Examples of bugs in embedded systems

Examples of bugs in embedded systems

- 1962 : NASA Mariner 1, Venus Probe (period instead of comma in FORTRAN DO loop)
- 1986/1987 : Therac-25 Incident radio-therapy device (patients died due to a bug in the control software)
- 1990 : AT&T long distance service fails for nine hours (wrong BREAK statement in C-code)
- 1994 : Pentium processor, division algorithm (incomplete entries in a look-up table)
- 1996 : Ariane 5, explosion (data conversion of a too large number)
- 1999 : Mars Climate Orbiters, Loss (Mixture of pound and kilograms)
- ...

Bug or feature ?

- most compilers use IEEE 754 floating point numbers
- but many of them have problems with that example in ANSI-C :

```
float q = 3.0/7.0;
if (q == 3.0/7.0) printf("no problem.");
else printf("problem!");
```

- try it, and you will see that C has a problem !
- reason : expressions in C computed in double precision, but the float q has only single precision
- no solution : avoid tests on equality
- instead, check if difference is very small :

```
float q = 3.0/7.0;
if fabs(q-3.0/7.0) <= epsilon
      printf("no problem");
else printf ("problem!");
```

- but this “equality” is no equivalence relation
- you may have x = y and y = z , but not x = z

Plan

Formal verification

One of the success stories of computer science

- Allows to verify large systems even systems with 10^{100} states
- But : requires a formal semantics of the system (mathematical model)
- Unfortunately, not available for most programming languages

Formal verification : aims

Formal verification : aims

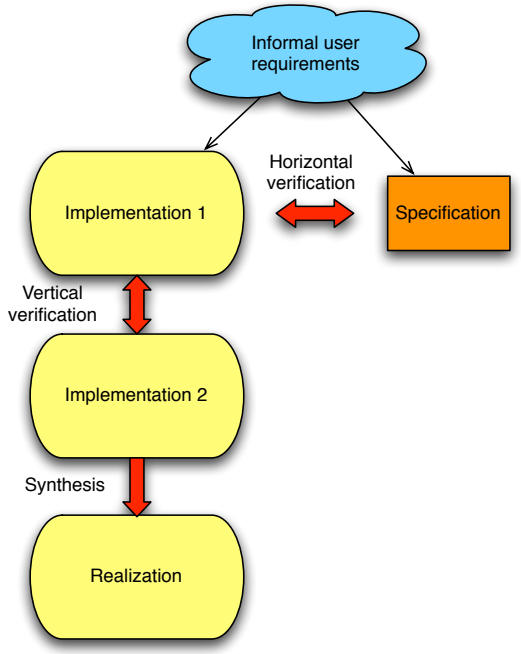
- Given a formal specification and a precise system description
- Check, whether the system satisfies the specification
- Done by generating some sort of mathematical proof
- can deal with
  - Correctness : no design errors
  - Reliability : system works all the time,
  - Security : no non-authorized usage,

Classes of faults

Classes of faults

- At specification : wrong/incomplete/vacuous specification
- Design errors : system does not satisfy the specification
- Faulty design tools : compiler generates wrong code
- Fabrication faults : faults on chips or other hardware parts

Horizontal vs. vertical verification



- Horizontal verification : system vs. property ;
- Vertical verification : system vs. refined system ;
- Synthesis : correctness preserving refinement.

Important Classes of Temporal Properties

Informal definition of some Temporal Properties

- Safety properties : unwanted system states are never reached
- Liveness properties : desired behavior eventually occurs
- Persistence properties : after some time, desired state set is never left
- Fairness properties : a request infinitely done is infinitely satisfied

Note : not all the specification logic can express all temporal properties (e.g. CTL can not express fairness).

Limits of Formal Verification

Limits of Formal Verification

- Was the specification right ?
  - Often given in natural language, thus imprecise
  - If formally given, often hard to read
  - Hard to validate : simulate/verify the specification ? against what ?
- Completeness of specification
  - Were all important properties specified ?

Two main approaches to formal verification

Model checking

- Systematically exhaustive exploration of the mathematical model
- Possible for finite models, but also for some infinite models where infinite sets of states can be effectively represented
- Usually consists of exploring all states and transitions in the model
- Efficient techniques
- If the model has a bug : provides counter-examples

Logical inference

- Mathematical reasoning, usually using theorem proving software (e.g. HOL or Isabelle theorem provers).
- Usually only partially automated and is driven by the user's understanding of the system to validate.

Plan

Historical facts on specification and verification

A few dates

- 1936 :Alan Turing defined his machine to reason about computability
- 1943 :McCulloch and Pitts used Finite State Automata to model neural cells
- 1956 : Kleene develops equivalence to regular expressions
- 1960 : Büchi develops ω-automata on infinite words
- 1962 : Carl Petri introduced the “Petri net” model ;
- 1963 : McCarthy ... : operational semantics : a computer program modeled as an execution of an abstract machine
- 1967-9 : Floyd, Hoare ... : axiomatic semantics : emphasis in proof methods. Program assertions, preconditions, postconditions, invariants.
- 1971 : Bekic : first idea of process algebra with a parallel operator
- 1971 : Scott, Strachey ... : denotational semantics : a computer program modeled as a function transforming input into output

A few dates (cont'd)

- 1973 : Park, de Bakker, de Roeve, least fixpoint operators
- 1976 : Edsger W. Dijkstra, notion of weakest preconditions (wps)
- 1977 : Amir Pnueli proposed using temporal logic for reasoning about computer program and defined LTL ;
- 1978 : C.A.R. (Tony) Hoare : book on the process algebra CSP
- 1980 : Robin Milner : book on the process algebra CCS
- 1980 : Edmund Clarke and Ellen Emerson defined the temporal logic CTL ;
- 1982 : Pratt and Kozen, μ-calculus
- 1985 : David Harel and Amir Pnueli used the term "reactive system" ;
- 1985 : Ellen Emerson, Chin-Laung Lei : defined the temporal logic CTL* ;
- 1986- : Symbolic model checking : BDD (Rendal Bryant), Partial order reduction (Antti Valmari, Patrice Godefroid)

A few dates (cont'd)

- 1988 : Ed Brinksma : defined LOTOS (process algebra with data)
- 1989 : Extended models of Process algebra (time, mobility, probabilities and stochastics, hybrid)
- 1989 : Rajeev Alur and David Dill : Timed automata
- 1995 : Rajeev Alur et al (Thomas Henzinger) : Hybrid automata
- 1990- : Huge research & developments

Turing awards in formal software development / verification

Turing Awards (From Wikipedia)

Often recognized as the "Nobel Prize of computing", the award is named after Alan Mathison Turing, a British mathematician who is "frequently credited for being the father of theoretical computer science and artificial intelligence".

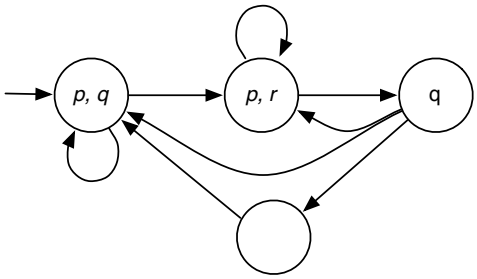
1972 : Edsger Dijkstra
1976 : Michael O. Rabin and Dana S. Scott
1978 : Robert W. Floyd
1980 : C. Antony R. Hoare
1991 : Robin Milner
1996 : Amir Pnueli
2007 : Edmund M. Clarke, E. Allen Emerson, and Joseph Sifakis
2030 : you ?

Chapter 2 : Kripke Structures and Labeled Transition Systems

Plan

Kripke Structure

- Transition system (behavior),
- Transitions are atomic actions,
- States are labeled with boolean variables that hold there (others are false),
- Computations are sequences of set of propositions corresponding to the states reached.



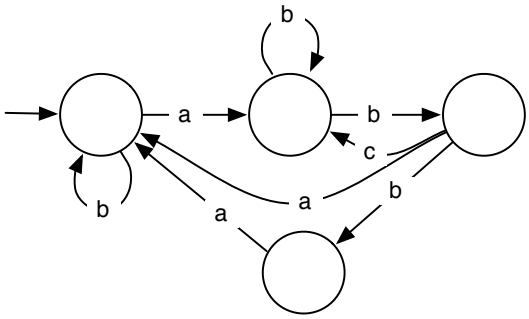
Definition : Kripke structure (Given $\mathcal{V}$: a set of propositions)

tuple $\mathcal{K} = (\mathcal{I}, \mathcal{S}, \mathcal{R}, \mathcal{L})$ with

- $\mathcal{S}$: the finite set of states
- $\mathcal{I} \subseteq \mathcal{S}$: the set of initial states
- $\mathcal{R} \subseteq \mathcal{S} \times \mathcal{S}$: the set of transitions (**Notation :** $s \underset{\mathcal{K}}{\rightarrow} s' \equiv (s, s') \in \mathcal{R}$)
- $\mathcal{L} : \mathcal{S} \rightarrow 2^{\mathcal{V}}$ the label function.

Labeled Transition System (LTS)

- Transition system : models a system's behavior,
- Transition are atomic actions,
- Transitions are labeled,
- Computations are sequences of labels corresponding to the transitions taken.



Given $\mathcal{L}$: a set of labels

Definition : Labeled Transition System

tuple $\mathcal{M} = (\mathcal{I}, \mathcal{S}, \mathcal{R})$ with

- $\mathcal{S}$: the finite set of states
- $\mathcal{I} \subseteq \mathcal{S}$: the set of initial states
- $\mathcal{R} \subseteq \mathcal{S} \times \mathcal{L} \times \mathcal{S}$: the set of transitions

In the first chapters we shall mainly work with Kripke structures

Computation Path

Path of a Kripke Structure

- An infinite path of a Kripke structure $\mathcal{K} = (\mathcal{I}, \mathcal{S}, \mathcal{R}, \mathcal{L})$ is a function $\pi : \mathbb{N} \to \mathcal{S}$ with $\pi^{(0)} \in \mathcal{I}$ and $\forall t. \pi^{(t)} \underset{\mathcal{K}}{\to} \pi^{(t+1)}$
- A path can be seen as an infinite sequence of states $\pi = s_0 s_1 s_2 \ldots$ such that $s_0 \in \mathcal{I}$ and $s_i \underset{\mathcal{K}}{\to} s_{i+1}$
- $Path_{\mathcal{K}}(s) = \{\pi \mid \pi^{(0)} = s \text{ and } \forall t. \pi^{(t)} \underset{\mathcal{K}}{\to} \pi^{(t+1)}\}$
- $Path_{\mathcal{K}}(S) = \bigcup_{s \in S} Path_{\mathcal{K}}(s)$
- Trace of a path π : sequence $\lambda t. \mathcal{L}(\pi^{(t)})$
- Language $Lang(s)$ of a state s : sequence $Lang(s) := \{\lambda t. \mathcal{L}(\pi^{(t)}) \mid \pi \in Paths_{\mathcal{K}}(s)\}$
- Language $Lang(\mathcal{K}) := \bigcup_{s \in \mathcal{I}} Lang(s)$

Computation Path

- In a Kripke Structure / LTS, the identifier of the states are not important at the semantical level.
- Given a path of a Kripke structure, the sequences of sets of variables which holds give its semantics
- [Given a path of a LTS, the sequences of labels of the transitions taken give its semantics]

Plan

Equivalence of Kripke structure / LTS

- Are $\mathcal{K}_1$ and $\mathcal{K}_2$ the same ?
- Depends on the classes of properties considered !
- We first give some preorder and equivalence relations

Equivalence of Kripke structure / LTS

- Obvious candidate : Isomorphism up to renaming of the states

Isomorphic structures

$\mathcal{K}_1 = (\mathcal{I}_1, \mathcal{S}_1, \mathcal{R}_1, \mathcal{L}_1)$ and $\mathcal{K}_2 = (\mathcal{I}_2, \mathcal{S}_2, \mathcal{R}_2, \mathcal{L}_2)$ are isomorphic, if there is a bijection $\Theta : \mathcal{S}_1 \mapsto \mathcal{S}_2$ with

- $\mathcal{S}_2 = \Theta(\mathcal{S}_1)$
- $s_1 \in \mathcal{I}_1 \iff \Theta(s_1) \in \mathcal{I}_2$
- $s_1 \underset{\mathcal{K}_1}{\rightarrow} s_2 \iff \Theta(s_1) \underset{\mathcal{K}_1}{\rightarrow} \Theta(s_2)$
- $\mathcal{L}_1(s_1) = \mathcal{L}_2(\Theta(s_1))$

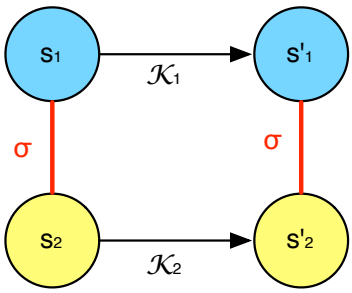
Generally much too strong equivalence !

Simulation relation

Simulation relation

Given $\mathcal{K}_1 = (\mathcal{I}_1, \mathcal{S}_1, \mathcal{R}_1, \mathcal{L}_1)$ and $\mathcal{K}_2 = (\mathcal{I}_2, \mathcal{S}_2, \mathcal{R}_2, \mathcal{L}_2)$, $\sigma \subseteq \mathcal{S}_1 \times \mathcal{S}_2$ is a simulation relation between $\mathcal{K}_1$ and $\mathcal{K}_2$ if the following holds :

1. $(s_1, s_2) \in \sigma$ implies $\mathcal{L}_1(s_1) = \mathcal{L}_2(s_2)$
2. $\forall (s_1, s_2) \in \sigma, \forall s_1'.s_1 \underset{\mathcal{K}_1}{\rightarrow} s_1', \exists s_2'.s_2 \underset{\mathcal{K}_2}{\rightarrow} s_2' \wedge (s_1', s_2') \in \sigma$
3. $\forall s_1 \in \mathcal{I}_1, \exists s_2 \in \mathcal{I}_2 : (s_1, s_2) \in \sigma$



$\mathcal{K}_1 \preccurlyeq^S \mathcal{K}_2$ (! ! $\mathcal{K}_2$ simulates $\mathcal{K}_1$! !)

- $\mathcal{K}_1 \preccurlyeq^S \mathcal{K}_2$:= there exists a simulation between $\mathcal{K}_1$ and $\mathcal{K}_2$
- $\preccurlyeq^S$ is a preorder
- $\mathcal{K}_1 \simeq^S \mathcal{K}_2 \Rightarrow \mathcal{K}_1 \preccurlyeq^S \mathcal{K}_2 \wedge \mathcal{K}_2 \preccurlyeq^S \mathcal{K}_1$ ($\mathcal{K}_1$ is similar to $\mathcal{K}_2$)
- $\simeq^S$ is an equivalence relation

Simulaton implies language inclusion

$\mathcal{K}_1 \preccurlyeq^S \mathcal{K}_2 \Rightarrow Lang(\mathcal{K}_1) \subseteq Lang(\mathcal{K}_2)$

Proof :

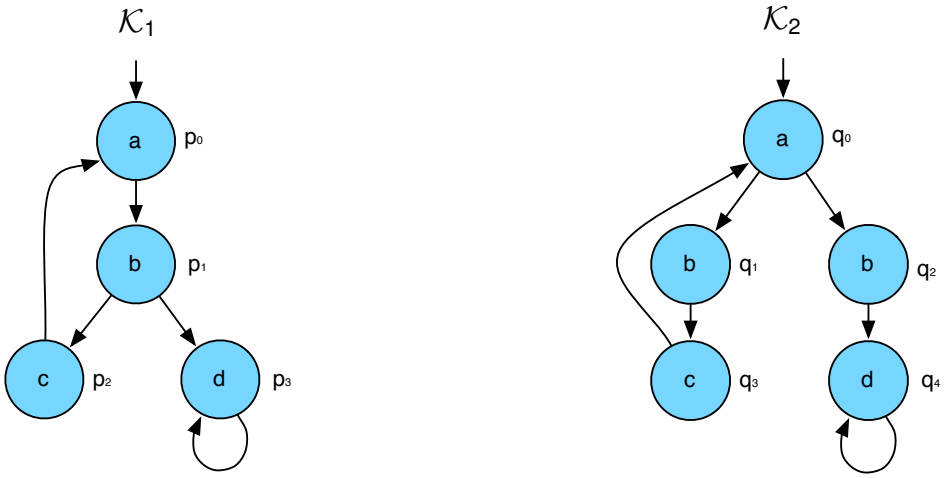
- We show that $\forall \omega \in Lang(\mathcal{K}_1) \Rightarrow \omega \in Lang(\mathcal{K}_2)$
- Let σ, the simulation relation in $\mathcal{S}_1 \times \mathcal{S}_2$ with $\forall s \in \mathcal{I}_1, \exists s' \in \mathcal{I}_2.(s, s') \in \sigma$
- Let π the path in $\mathcal{K}_1$ with trace ω
- There is a corresponding path π' in $\mathcal{K}_2$ with
- $\forall i \in \mathbb{N}.(\pi_i, \pi_i') \in \sigma$ (by induction)
- Hence $\forall i \in \mathbb{N}.\mathcal{L}(\pi_i) = \mathcal{L}(\pi_i')$
- which concludes the proof.

Checking simulation preorder

Algorithm to check simulation

- $(s_1, s_2) \in \mathcal{H}_0 \iff \mathcal{L}_1(s_1) = \mathcal{L}_2(s_2)$
- $(s_1, s_2) \in \mathcal{H}_{i+1} \iff \left(\begin{array}{l} (s_1, s_2) \in \mathcal{H}_i \wedge \\ \forall s_1' \in \mathcal{S}_1.\ s_1 \underset{\mathcal{K}_1}{\rightarrow} s_1' \exists s_2' \in \mathcal{S}_2. \\ \quad s_2 \underset{\mathcal{K}_2}{\rightarrow} s_2' \ \wedge \ (s_1', s_2') \in \mathcal{H}_i \end{array} \right)$
- Until stabilization ($\mathcal{H}_{i+1} = \mathcal{H}_i$)

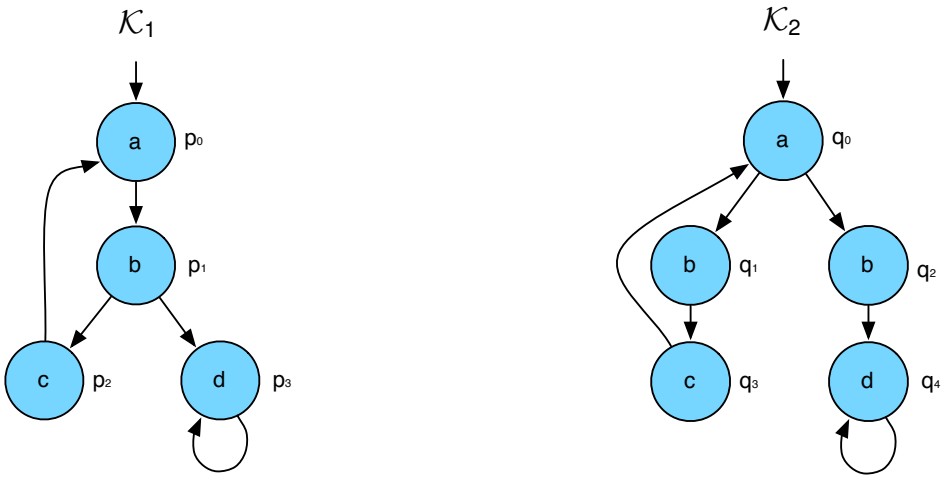
Computing the greatest simulation relation between $\mathcal{K}_1$ and $\mathcal{K}_2$



- $\mathcal{H}_0 = \{(p_0, q_0), (p_1, q_1), (p_1, q_2), (p_2, q_3), (p_3, q_4)\}$
- $\mathcal{H}_1 = \{(p_0, q_0), , (p_2, q_3), (p_3, q_4)\}$
- $\mathcal{H}_2 = \{(p_2, q_3), (p_3, q_4)\}$
- $\mathcal{H}_3 = \{(p_3, q_4)\}$

$\Rightarrow$ no state related to p_0 : $\mathcal{K}_1 \not\preceq^S \mathcal{K}_2$

Computing the greatest simulation between $\mathcal{K}_2$ and $\mathcal{K}_1$



- $\mathcal{H}_0 = \{(q_0, p_0), (q_1, p_1), (q_2, p_1), (q_3, p_2), (q_4, p_3)\}$

$\Rightarrow$ stable and $(q_0, p_0) \in \mathcal{H} : \mathcal{K}_2 \preccurlyeq^S \mathcal{K}_1$

Equivalence of Kripke structure / LTS

- Other obvious candidate : Language equivalence

Algorithm to test language inclusion ($\mathcal{K}_1 \preccurlyeq^L \mathcal{K}_2$)

1. Determinisation of $\mathcal{K}_2$: $\mathcal{K}_2'$
2. Check if $\mathcal{K}_1 \preccurlyeq^S \mathcal{K}_2'$

- $\mathcal{K}_1 \simeq^L \mathcal{K}_2 \Leftrightarrow \mathcal{K}_1 \preccurlyeq^L \mathcal{K}_2 \wedge \mathcal{K}_2 \preccurlyeq^L \mathcal{K}_1$
- But determinisation of Kripke structure / LTS (as finite automata) is hard
- computing simulation or bisimulation is more efficient

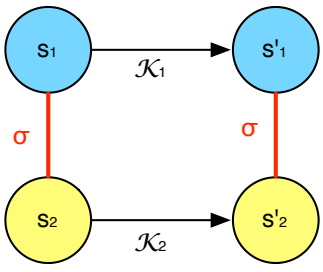
Bisimulation relation

Bisimulation relation

Given $\mathcal{K}_1 = (\mathcal{I}_1, \mathcal{S}_1, \mathcal{R}_1, \mathcal{L}_1)$ and $\mathcal{K}_2 = (\mathcal{I}_2, \mathcal{S}_2, \mathcal{R}_2, \mathcal{L}_2)$, $\sigma \subseteq \mathcal{S}_1 \times \mathcal{S}_2$ is a bisimulation relation between $\mathcal{K}_1$ and $\mathcal{K}_2$ ($\mathcal{K}_1 \simeq^B_\sigma \mathcal{K}_2$) if the following holds :

1. $(s_1, s_2) \in \sigma$ implies $\mathcal{L}_1(s_1) = \mathcal{L}_2(s_2)$
2. $\forall (s_1, s_2) \in \sigma, \forall s'_1 . s_1 \underset{\mathcal{K}_1}{\rightarrow} s'_1, \exists s'_2 . s_2 \underset{\mathcal{K}_2}{\rightarrow} s'_2 \wedge (s'_1, s'_2) \in \sigma$
3. $\forall (s_1, s_2) \in \sigma, s_2 \underset{\mathcal{K}_2}{\rightarrow} s'_2, \exists s'_1 . s_1 \underset{\mathcal{K}_1}{\rightarrow} s'_1 \wedge (s'_1, s'_2) \in \sigma$
4. $\forall s_1 \in \mathcal{I}_1, \exists s_2 \in \mathcal{I}_2 : (s_1, s_2) \in \sigma$
5. $\forall s_2 \in \mathcal{I}_2, \exists s_1 \in \mathcal{I}_1 : (s_1, s_2) \in \sigma$

Bisimulation relation



$\mathcal{K}_1 \simeq^B \mathcal{K}_2$

- if there exists a bisimulation σ between $\mathcal{K}_1$ and $\mathcal{K}_2$
- $\mathcal{K}_1 \simeq^B \mathcal{K}_2 \Rightarrow \mathcal{K}_1 \simeq^S \mathcal{K}_2$
- $\simeq^B$ is an equivalence

Checking bisimulation

First idea

Algorithm similar to the one presented for simulation (but check both sides at each step)

A more efficient method due to Paige & Tarjan exists

Checking bisimulation

Basic definitions / notations

- The following notations are used
  - I : an index set
  - ρ : a partition of the set of states $\rho = \{B_i \mid i \in I\}$
  - $R[x], R[X]$ for a state x (resp. a set of states X), is the set of states that are successors of x (resp. X)
- ρ' refines ρ $\underline{\text{iff}}$ $\forall B' \in \rho', \exists B \in \rho \mid B' \subseteq B$
  (notation $\rho' \sqsubseteq \rho$)

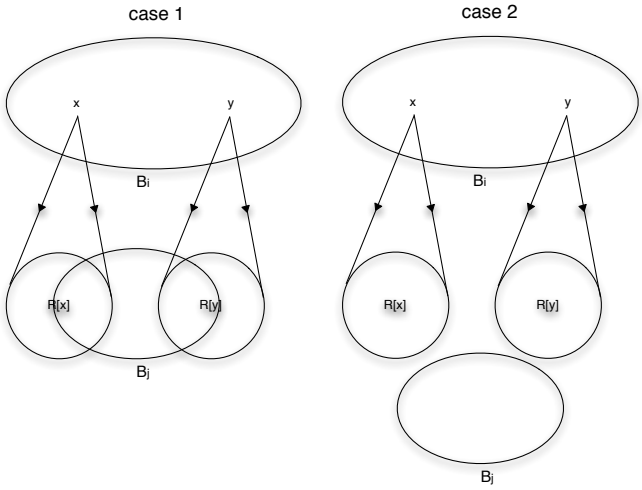
A partition ρ is compatible with the relation R

- $\underline{\text{iff}}$ $\forall i \in I, \forall x, y \in B_i, \mathcal{L}(x) = \mathcal{L}(y) \wedge$
  $R[x] \cap B_j \neq \Phi \Leftrightarrow R[y] \cap B_j \neq \Phi$
- $\underline{\text{iff}}$ $\forall i \in I, \forall x, y \in B_i, \mathcal{L}(x) = \mathcal{L}(y) \wedge$
  $\forall B, B' \in \rho,$ either $B' \subseteq R^{-1}[B],$ or $B' \cap R^{-1}[B] = \Phi$

ρ is compatible with the relation R : first criteria

A partition ρ is compatible with the relation R iff

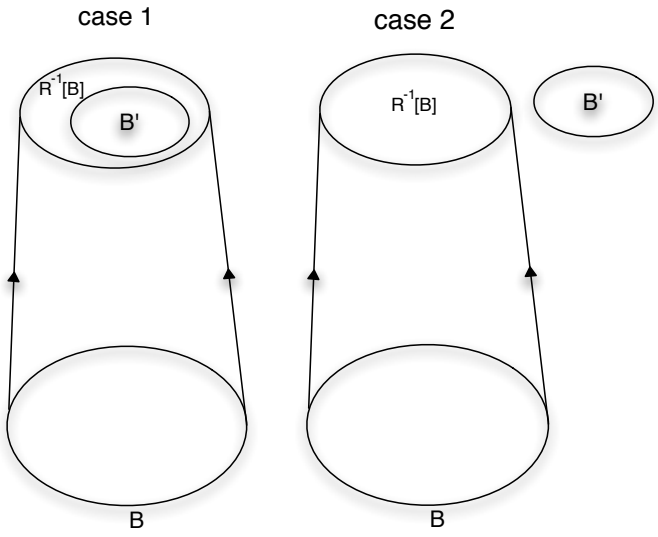
- $\forall i \in I, \forall x, y \in B_i, \mathcal{L}(x) = \mathcal{L}(y) \wedge$
  $R[x] \cap B_j \neq \Phi \Leftrightarrow R[y] \cap B_j \neq \Phi$



ρ is compatible with the relation R : second criteria

A partition ρ is compatible with the relation R

- iff $\forall i \in I, \forall x, y \in B_i, \mathcal{L}(x) = \mathcal{L}(y) \ \wedge \ \forall B, B' \in \rho,$
  either $B' \subseteq R^{-1}[B]$, or $B' \cap R^{-1}[B] = \Phi$



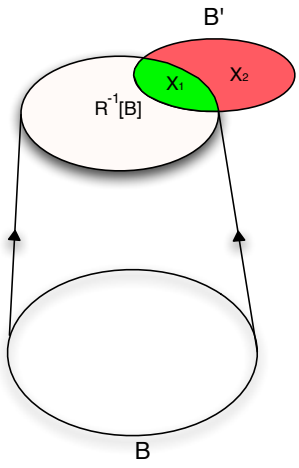
Checking bisimulation

Proposition (link between bisimulation and compatibility)

ρ is a bisimulation iff ρ is compatible with R.

Algorithm to compute the corsest partition compatible with R

- Start from the partition $\rho = \{B \mid \forall x, y \in B.Lang(x) = Lang(y)\}$
- and calculate the coarsest equivalence relation ρ' compatible with R and which refines ρ
- **Principle :** refine B' by
  $X_1 = B' \cap R^{-1}[B]$ and
  $X_2 = B' \setminus R^{-1}[B]$



Simple Algorithm to compute the coarset partition compatible with *R*

```
{
  r := {B | forall x,y in B.
       Lang(x) = Lang(y)};   // classes
  W := r                     // splitter
  while W not empty do
  {
    select B in W;
    suppress B in W;
    call Interpred(R-1[B]); // Compute a set og (B',X1)
                            // with all classes B' in r
                            // incompatible with R-1[B]
                            // X1 is the resulting intersect.
```


Simple Algorithm to compute the coarset partition compatible with *R* (2)

```
    for all (B',X1) in interpred do
    {
      X2 = B'\X1;
      replace B' by X1 and X2 in r;
      if B' in W then
         replace B' by X1 and X2 in W;
      else
          add X1 and X2 in W;
      fi
    }
  }
}
```

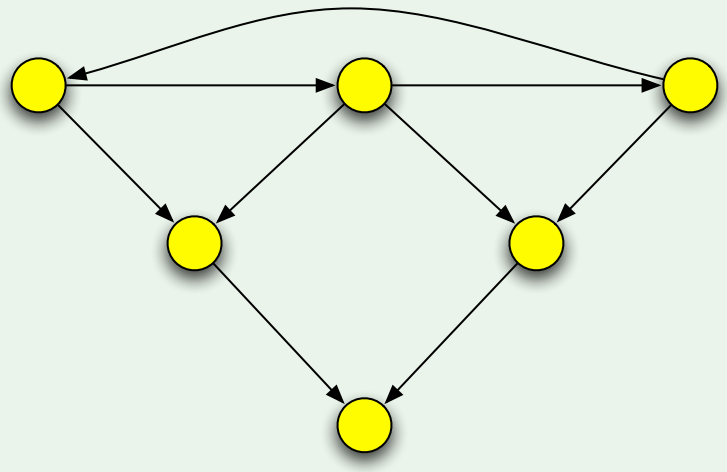
Simple Algorithm to compute the coarset partition compatible with *R* (2)

```
procedure interpred(X)
{
  interpred := empty;
  for all B' in r do
  {
    X1 := B' intersection X;
    if X1 not empty and X1 <> B' then
      interpred := interpred Union {(B',X1)};
    fi
  }
}
```


Paige-Tarjan's algorithm on an example

Paige-Tarjan's algorithm on an example

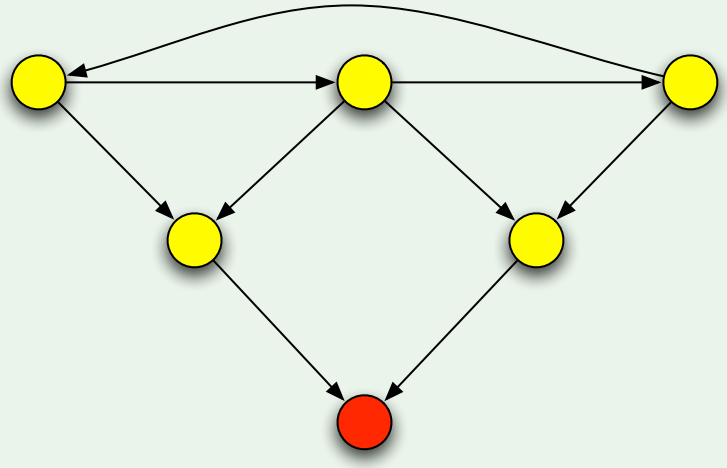
Suppose all states have the same set of propositions



Paige-Tarjan's algorithm on an example

Paige-Tarjan's algorithm on an example

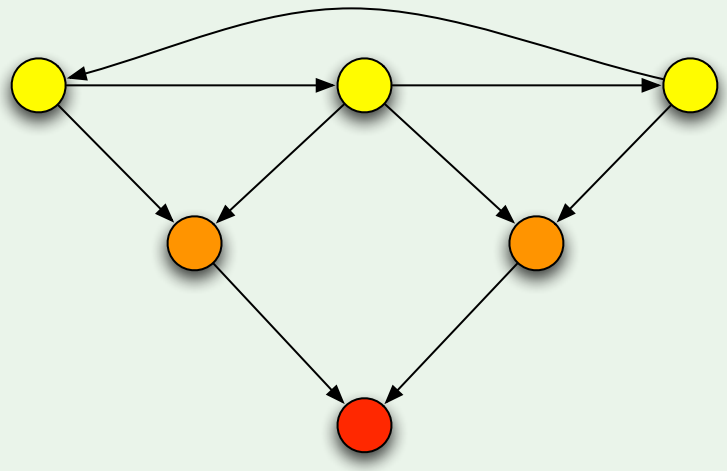
Suppose all states have the same set of propositions



Paige-Tarjan's algorithm on an example

Paige-Tarjan's algorithm on an example

Suppose all states have the same set of propositions



Optimized version of Paige-Tarjan's algorithm

Complexity of Paige-Tarjan's algorithm

The optimized version of the Paige-Tarjan's algorithm (with a clever way to handle splitters) has a complexity $\mathcal{O}(n \log n)$

Paige-Tarjan's algorithm for LTS

Principle of Paige-Tarjan's algorithm for LTS

- The first partition is the set of states
- The refinement process uses $R_a^{-1}[B]$ for all label a

Plan

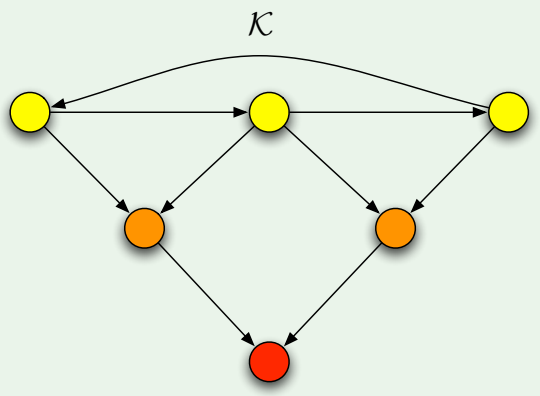
Quotient structures

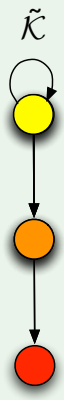
Quotient structures

- Given $\mathcal{K} = (\mathcal{I}, \mathcal{S}, \mathcal{R}, \mathcal{L})$
- Given an equivalence relation $\sigma \subseteq \mathcal{S} \times \mathcal{S}$
- with $(s_1, s_2) \in \sigma \Rightarrow \mathcal{L}(s_1) = \mathcal{L}(s_2)$
- The quotient structure of $\mathcal{K}$ for σ : $\mathcal{K}_{/\sigma} = (\tilde{\mathcal{I}}, \tilde{\mathcal{S}}, \tilde{\mathcal{R}}, \tilde{\mathcal{L}})$ =
  - $\tilde{\mathcal{I}} := \{\{s' \in \mathcal{S} \mid (s, s') \in \sigma\} \mid s \in \mathcal{I}\}$
  - $\tilde{\mathcal{S}} := \{\{s' \in \mathcal{S} \mid (s, s') \in \sigma\} \mid s \in \mathcal{S}\}$
  - $(\tilde{s_1}, \tilde{s_2}) \in \tilde{\mathcal{R}} \iff \forall s'_1 \in \tilde{s_1} \exists s'_2 \in \tilde{s_2}.(s'_1, s'_2) \in \mathcal{R}$
  - $\tilde{\mathcal{L}}(\tilde{s}) := \mathcal{L}(s)$

Quotient structure

Quotient structure : example





- A state of $\mathcal{K}_{/\sigma}$ is an equivalence class of states of $\mathcal{K}$ for σ
- Depending on the relation considered, $\mathcal{K}_{/\sigma}$ preserves various classes of properties
- Bisimilarity preserves most of the properties (see next chapters)

Product of Kripke structures

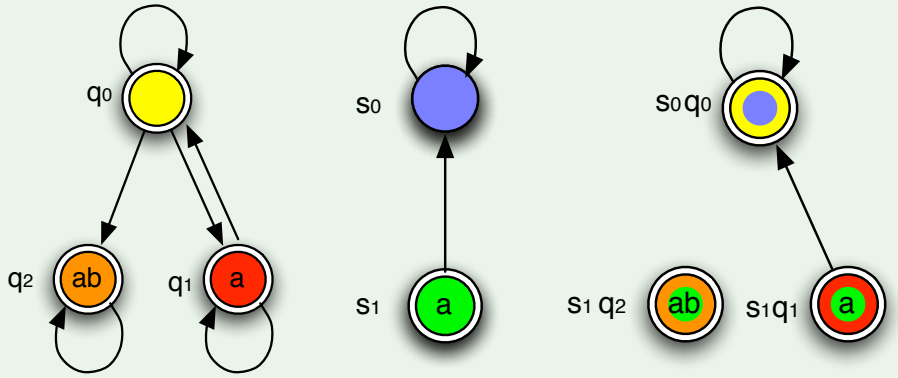
Product $\mathcal{K}_1 \times \mathcal{K}_2$

- Given $\mathcal{K}_1 = (\mathcal{I}_1, \mathcal{S}_1, \mathcal{R}_1, \mathcal{L}_1)$ and $\mathcal{K}_2 = (\mathcal{I}_2, \mathcal{S}_2, \mathcal{R}_2, \mathcal{L}_2)$ over resp. $\mathcal{V}_1$ and $\mathcal{V}_2$
- $\mathcal{K}_1 \times \mathcal{K}_2 = (\mathcal{I}_\times, \mathcal{S}_\times, \mathcal{R}_\times, \mathcal{L}_\times)$ over variables $\mathcal{V}_1 \cup \mathcal{V}_2$
  - $\mathcal{S}_\times := \{(s_1, s_2) \in \mathcal{S}_1 \times \mathcal{S}_2 \mid \mathcal{L}_1(s_1) \cap \mathcal{V}_2 = \mathcal{L}_2(s_2) \cap \mathcal{V}_1)\}$
  - $\mathcal{I}_\times := \mathcal{S}_\times \cap (\mathcal{I}_1 \times \mathcal{I}_2)$
  - $\mathcal{R}_\times := \{((s_1, s_2), (s_1', s_2')) \in \mathcal{S}_\times \times \mathcal{S}_\times \mid (s_1, s_1') \in \mathcal{R}_1 \wedge (s_2, s_2') \in \mathcal{R}_2$
  - $\mathcal{L}_\times(s_1, s_2)) := \mathcal{L}_1(s_1) \cup \mathcal{L}_2(s_2))$

- $\mathcal{K}_1 \times \mathcal{K}_2$ models synchronous parallel executions
- $\mathcal{K}_1 \times \mathcal{K}_2$ contains only paths that appear in $\mathcal{K}_1$ and $\mathcal{K}_2$
- may have no states !
- may have no transitions !

Product of Kripke Structure

$\mathcal{K}_\times = \mathcal{K}_1 \times \mathcal{K}_2$



Existential and universal predecessors and successors

Existential and universal predecessors and successors

Given a relation $\mathcal{R} \subseteq \mathcal{S}_1 \times \mathcal{S}_2$, we define

- $\mathrm{pre}^{\mathcal{R}}_{\exists}(Q_2) := \{s_1 \in \mathcal{S}_1 \mid \exists s_2.(s_1, s_2) \in \mathcal{R} \wedge s_2 \in Q_2\}$
- $\mathrm{pre}^{\mathcal{R}}_{\forall}(Q_2) := \{s_1 \in \mathcal{S}_1 \mid \forall s_2.(s_1, s_2) \in \mathcal{R} \rightarrow s_2 \in Q_2\}$
- $\mathrm{suc}^{\mathcal{R}}_{\exists}(Q_1) := \{s_2 \in \mathcal{S}_2 \mid \exists s_1.(s_1, s_2) \in \mathcal{R} \wedge s_1 \in Q_1\}$
- $\mathrm{suc}^{\mathcal{R}}_{\forall}(Q_1) := \{s_2 \in \mathcal{S}_2 \mid \forall s_1.(s_1, s_2) \in \mathcal{R} \rightarrow s_1 \in Q_1\}$

- $\mathrm{pre}^{\mathcal{R}}_{\exists}(Q_2) :=$ the set of states that have a successor in Q_2
- $\mathrm{pre}^{\mathcal{R}}_{\forall}(Q_2) :=$ the set of states that have no successor in $\mathcal{S} \setminus Q_2$

Important properties of predecessors and successors

Important properties of predecessors and successors

Duality laws

- $\mathrm{pre}^{\mathcal{R}}_{\exists}(Q_2) := \mathcal{S}_1 \setminus \mathrm{pre}^{\mathcal{R}}_{\forall}(S_2 \setminus Q_2)$
- $\mathrm{pre}^{\mathcal{R}}_{\forall}(Q_2) := \mathcal{S}_1 \setminus \mathrm{pre}^{\mathcal{R}}_{\exists}(S_2 \setminus Q_2)$
- $\mathrm{suc}^{\mathcal{R}}_{\exists}(Q_1) := \mathcal{S}_2 \setminus \mathrm{suc}^{\mathcal{R}}_{\forall}(\mathcal{S}_1 \setminus Q_1)$
- $\mathrm{suc}^{\mathcal{R}}_{\forall}(Q_1) := \mathcal{S}_2 \setminus \mathrm{suc}^{\mathcal{R}}_{\exists}(\mathcal{S}_1 \setminus Q_1)$

Monotonicity laws

all these functions are motonic : e.g.

$$Q_2 \subseteq Q_2' \Rightarrow \mathrm{pre}^{\mathcal{R}}_{\exists}(Q_2) \subseteq \mathrm{pre}^{\mathcal{R}}_{\exists}(Q_2')$$

Plan

Big picture of Verification, Model Checking and Testing

Given :

$\mathcal{M}$: the (model) of the system developped

$\mathcal{S}$: the (model) of the (correct) system to provide

Φ : a specification of a required property

$\mathcal{T}$: a set of correct behaviors (i.e. $[\![\mathcal{T}]\!] \subseteq [\![\mathcal{S}]\!]$)

1. Verification checks that $[\![\mathcal{M}]\!] \simeq [\![\mathcal{S}]\!]$
2. Model checking checks that $[\![\mathcal{M}]\!] \subseteq [\![\Phi]\!]$
3. Testing checks that $[\![\mathcal{T}]\!] \subseteq [\![\mathcal{M}]\!]$

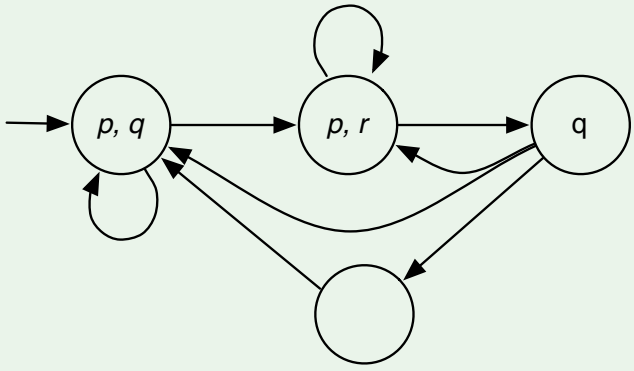
- Generally we do not have $\mathcal{S}$
- Verification is difficult
- Model checking may forget important properties and therefore does not provide a full verification
- Testing can show that the system has a bug but cannot prove that it is fully correct.

Chapter 3 : Temporal logics

Plan

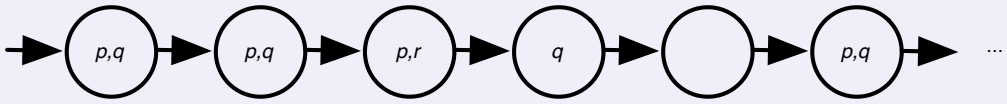
Example of linear vs branching semantics

Given the Kriepke structure



Linear time

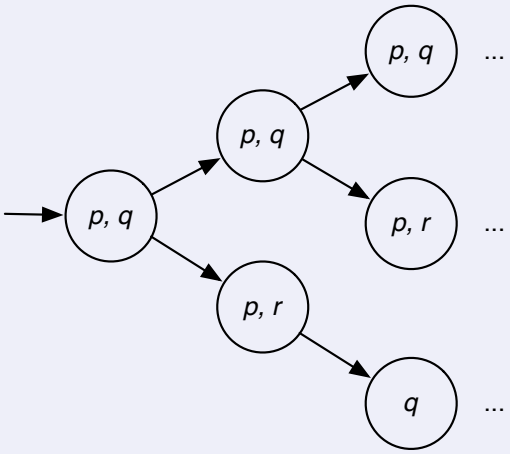
Trace semantics



- $[\![S]\!] = \{$ traces of $S\}$

Branching time

Computation tree semantics



- $[\![S]\!]$ = tree given by the unfolding of S

Choice Linear vs Branching

Closed vs. open system ?

- Closed systems : complete / self-sufficient
- Open systems : interact with the (unknown) environment.

Example : Controller C of an industrial equipment E :

- If E can be modeled : closed system $S \equiv C \times E$
- If E is unknown : open system $S \equiv C$ and E is the environment

Open vs. closed systems

Kripke structures vs. Labeled Transition Systems

- Kripke structures generally model global states of closed systems ;
- Labeled Transition Systems (LTS) generally model possible interactions of open systems with environments.

Linear vs. branching time logics

Depends also on the properties to check and the complexity of the algorithms

- E.g. : for reachability properties, trace semantics is enough.

Plan

Model checking $S \models \phi$ (S is a valid model for ϕ)

Model checking in linear semantics

- $[\![S]\!]$ = set of traces S can have

$\Rightarrow \forall \sigma \in [\![S]\!].\sigma \models \phi$

- with $[\![\phi]\!] = \{\sigma \mid \sigma \models \phi\}$ (set of valid traces)

$\Rightarrow [\![S]\!] \subseteq [\![\phi]\!]$

Model checking in branching semantics

- $[\![S]\!]$ = computation tree of S
- with $[\![\phi]\!] = \{Tree \mid Tree \models \phi\}$ (set of valid trees)

$\Rightarrow [\![S]\!] \in [\![\phi]\!]$

Plan

Informal presentation

- First introduced by Amir Pnueli in 1977 :
- Defines formulae which are valuated on infinite paths ;
- Uses temporal operators ;
- Therefore LTL is a Linear (time) temporal logic ;
- The semantics of a system is given by the set of paths it can have.

LTL syntax

LTL syntax

Given a set of propositions $\mathcal{P}$, a formula in Linear Temporal Logic (LTL) is defined using the following grammar :

$$\phi ::= \top \mid \bot \mid \quad p \mid \neg\phi \mid \phi \vee \phi \mid \phi \wedge \phi \mid \bigcirc \phi \mid \phi \,\mathrm{U}\, \phi \mid \phi \,\tilde{\mathrm{U}}\, \phi$$

where $p \in \mathcal{P}$.

- Most of the operators are standard ;
- $\bigcirc$ is the next operator ; $\bigcirc\phi$ is true if ϕ is true after the first state of the path ;
- U is the until operator ; $\phi \mathrm{U} \psi$ is true if ϕ is true in the path in all the states preceding one state where ψ is true.
- $\tilde{\mathrm{U}}$ is the release operator ; $\phi \tilde{\mathrm{U}} \psi$ is true if ψ is always true in the path unless this obligation is released by ϕ being true in a previous state.

LTL syntax

LTL syntax

Given a set of propositions $\mathcal{P}$, a formula in Linear Temporal Logic (LTL) is defined using the following grammar :

$$\phi ::= \top \mid \bot \mid \quad p \mid \neg\phi \mid \phi \vee \phi \mid \phi \wedge \phi \mid \bigcirc \phi \mid \phi \,\mathrm{U}\, \phi \mid \phi \,\tilde{\mathrm{U}}\, \phi$$

where $p \in \mathcal{P}$.

Derived operators

- $\Diamond \phi \equiv \top \,\mathrm{U}\, \phi$ (finally).
- $\Box \phi \equiv \neg\Diamond \neg\phi \equiv \bot \,\tilde{\mathrm{U}}\, \phi$ (globally).

Semantics of LTL

Semantics of LTL (on traces $\sigma = s_0 s_1 s_2 \ldots$)

With $\sigma = \sigma_0 \quad \sigma_i = s_i s_{i+1} \ldots$

$$
\begin{array}{lll}
\sigma \models \top & & \\
\sigma \not\models \bot & & \\
\sigma \models p & \underline{\text{iff}} & p \in \mathcal{L}(s_0) \\
\sigma \models \neg\phi & \underline{\text{iff}} & \sigma \not\models \phi \\
\sigma \models \phi_1 \vee \phi_2 & \underline{\text{iff}} & \sigma \models \phi_1 \vee \sigma \models \phi_2 \\
\sigma \models \phi_1 \wedge \phi_2 & \underline{\text{iff}} & \sigma \models \phi_1 \wedge \sigma \models \phi_2 \\
\sigma \models \bigcirc\phi & \underline{\text{iff}} & \sigma_1 \models \phi \\
\sigma \models \phi_1 \mathrm{U} \phi_2 & \underline{\text{iff}} & \exists i . \sigma_i \models \phi_2 \wedge \forall 0 \leq j < i . \sigma_j \models \phi_1 \\
\sigma \models \phi_1 \tilde{\mathrm{U}} \phi_2 & \underline{\text{iff}} & \forall i \geq 0 . \sigma_i \not\models \phi_2 \rightarrow \exists 0 \leq j < i . \sigma_j \models \phi_1
\end{array}
$$

Ltl with negation only on propositions

We can restrict definition of LTL with negation only applied to atomic proposition

restricted LTL syntax

Given a set of propositions $\mathcal{P}$, a formula in Linear Temporal Logic (LTL) is defined using the following grammar :

$$\phi ::= \top \mid \bot \mid \quad p \mid \neg p \mid \phi \vee \phi \mid \phi \wedge \phi \mid \bigcirc \phi \mid \phi \,\mathrm{U}\, \phi \mid \phi \,\tilde{\mathrm{U}}\, \phi$$

where $p \in \mathcal{P}$.

Translation extended Ltl to restricted Ltl

- $\neg(\phi_1 \,\mathrm{U}\, \phi_2) \equiv (\neg\phi_1) \,\tilde{\mathrm{U}}\, (\neg\phi_2)$
- $\neg(\phi_1 \,\tilde{\mathrm{U}}\, \phi_2) \equiv (\neg\phi_1) \,\mathrm{U}\, (\neg\phi_2)$
- $\neg \bigcirc \phi_1 \equiv \bigcirc \neg\phi_1$

LTL model checking

Principle

- $S \models \phi \equiv [\![S]\!] \subseteq [\![\phi]\!] \equiv [\![S]\!] \cap [\![\neg\phi]\!] = \varnothing$
- Since $[\![S]\!]$ and $[\![\phi]\!]$ are infinite sets, the idea is to work with automata
- S is a Kripke structure ✓
- For $\neg\phi$? : Büchi automata (see chapter 4)
- The algorithm will be given in Chapter 6).

Plan

Informal presentation

- First introduced by Allen Emerson and Edmund Clarke in 1981 ;
- Defines formulae which are valuated on infinite trees ;
- Uses temporal operators ;
- Therefore CTL is a branching (time) temporal logic ;

CTL syntax

CTL syntax

Given a set of propositions $\mathcal{P}$, a formula in Computation Tree Logic (CTL) is defined using the following grammar :

$$\phi ::= \top \mid p \mid \neg\phi \mid \phi \vee \phi \mid \exists \bigcirc \phi \mid \forall \bigcirc \phi \mid \exists \phi \, \mathrm{U} \, \phi \mid \forall \phi \, \mathrm{U} \, \phi$$

where $p \in \mathcal{P}$.

- Most of the operators are standard ;
- $\exists \bigcirc$ is the exists next operator ; $\exists \bigcirc \phi$ is true if there is a path (from the current state) where ϕ is true after the first state of the path ;
- U is the until operator ; $\exists \phi \mathrm{U} \psi$ is true if there is a path where ϕ is true in all the states preceding one state where ψ is true.
- $\forall \bigcirc$ and $\forall \mathrm{U}$ are similar but for all paths.

Semantics of CTL

Semantics of CTL (on tree and uses traces $\sigma = s_0 s_1 s_2 \ldots$)

$$
\begin{array}{lll}
s_0 \models \top & & \\
s_0 \models p & \underline{\text{iff}} & p \in \mathcal{L}(s_0) \\
s_0 \models \neg\phi & \underline{\text{iff}} & s_0 \not\models \phi \\
s_0 \models \phi_1 \vee \phi_2 & \underline{\text{iff}} & s_0 \models \phi_1 \vee s_0 \models \phi_2 \\
s_0 \models \exists \bigcirc \phi & \underline{\text{iff}} & \exists s_1 . s_0 \underset{\mathcal{K}}{\rightarrow} s_1 \wedge s_1 \models \phi \\
s_0 \models \forall \bigcirc \phi & \underline{\text{iff}} & \forall s_1 . s_0 \underset{\mathcal{K}}{\rightarrow} s_1 \rightarrow s_1 \models \phi \\
s_0 \models \exists \phi_1 \mathrm{U} \phi_2 & \underline{\text{iff}} & \exists \sigma \in \mathit{Path}(s_0) . \exists i . \sigma^i \models \phi_2 \wedge \forall 0 \leq j < i . \sigma^j \models \phi_1 \\
s_0 \models \forall \phi_1 \mathrm{U} \phi_2 & \underline{\text{iff}} & \forall \sigma \in \mathit{Path}(s_0) . \exists i . \sigma^i \models \phi_2 \wedge \forall 0 \leq j < i . \sigma^j \models \phi_1
\end{array}
$$

CTL syntax (cont'd)

Derived operators

- $\exists \Diamond \phi \equiv \exists \top \, \mathrm{U} \, \phi$ (exists finally).
- $\forall \Diamond \phi \equiv \forall \top \, \mathrm{U} \, \phi$ (for all finally).
- $\exists \Box \phi \equiv \neg \forall \Diamond \neg \phi$ (exists globally).
- $\forall \Box \phi \equiv \neg \exists \Diamond \neg \phi$ (for all globally).

CTL without universal quantifiers (is sometimes useful)

In every CTL formula, every universal quantifiers can be replaced using the following equivalence :

- $\forall \bigcirc \phi \equiv \neg \exists \bigcirc \neg \phi$
- $\forall \phi_1 \, \mathrm{U} \, \phi_2 \equiv \neg \exists (\neg \phi_2 \, \mathrm{U} \, (\neg \phi_1 \wedge \neg \phi_2)) \wedge \neg \exists (\Box \neg \phi_2)$
- $\forall \Box \phi \equiv \neg \exists \Diamond \neg \phi$
- $\forall \Diamond \phi \equiv \neg \exists \Box \neg \phi$

CTL model checking

Principle

- For every state s in $\mathcal{K}$, decorate s with all the subformulae ϕ_i of ϕ such that $s \models \phi_i$
- More efficient algorithms use a translation of CTL formulae into μ-calculus formulae (see chapter 5) and a symbolic model checking algorithm (see chapter 6).

Chapter 4 : ω-automata

Plan

Need to extend Finite automata

Finite and infinite words

- Given an alphabet Σ
- Σ^* is the set of words of finite length
- an infinite word w is defined by a mapping $w : \mathbb{N} \mapsto \Sigma$

Need to extend Finite automata

- In the 50s, finite automata define languages on finite words.
- Need a formalism to define languages on infinite words
- Julius Richard Büchi studied the problem $\Rightarrow$ Büchi automata
- We will see the link between Büchi automata and LTL

Plan

Büchi automaton

$\mathcal{A} = (\Sigma, \mathcal{Q}, \mathcal{I}, \mathcal{R}, \mathcal{F})$ where

- $\Sigma = \{\Sigma_1, \Sigma_2, \ldots, \Sigma_m\}$ is the set of input symbols
- $\mathcal{Q} = \{q_1, q_2, \ldots, q_n\}$ is the set of states
- $\mathcal{I} \subseteq \mathcal{Q}$ is the set of initial states
- $\mathcal{R} \subseteq \mathcal{Q} \times \sigma \times \mathcal{Q}$ is the transition relation
- $\mathcal{F} \subseteq \mathcal{Q}$ is the set of accepting states

Deterministic Büchi automaton

- $\mathcal{I} = \{q_1\}$
- $|\{q' \mid \exists q \in \mathcal{Q}, \sigma \in \Sigma.(q, \sigma, q') \in \mathcal{R}\}| = 1$

Buchi acceptance condition

$inf(\sigma)$

Given an infinite sequence of states σ. $inf(\sigma)$ is the set of states that appear infinitely often in σ

w accepted by $\mathcal{A}$

An infinite word w is accepted by the automaton $\mathcal{A}$ if there exists an infinite path ρ : $\rho : \mathbb{N} \mapsto \mathcal{Q}$ such that

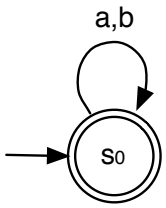
- $\rho(0) \in \mathcal{I}$ (the path starts at an initial state)
- $\forall i \in \mathbb{N}.(\rho(i), w_i, \rho(i+1)) \in \mathcal{R}$
- $\underline{\inf}(\rho) \cap \mathcal{F} \neq \varnothing$

Informally, $\mathcal{A}$ accepts w with a path which runs infinitely often through an accepting state.

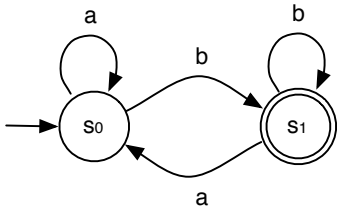
$\mathcal{L}(\mathcal{A})$

$\mathcal{L}(\mathcal{A}) = \{w \mid w \text{ accepted by } \mathcal{A}\}$

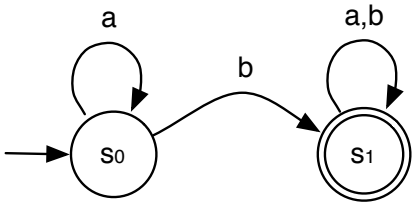
Examples of Büchi automata



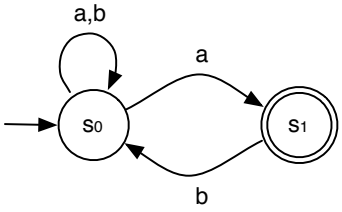
$\mathcal{L}(\mathcal{A}_1) = (a+b)^\omega$



$\mathcal{L}(\mathcal{A}_3) = a^*b(b+aa^*b)^\omega = a^*(ba^*)^\omega$



$\mathcal{L}(\mathcal{A}_2) = a^*b(a+b)^\omega$



$\mathcal{L}(\mathcal{A}_4) = (a+b)^*a(b(a+b)^*a)^\omega = ((a+b)^*ab)^\omega$

Other types of acceptance conditions

Other types of acceptance conditions

- Büchi : $\mathcal{F} \subseteq \mathcal{Q}$,
  $\underline{\inf}(\rho) \cap \mathcal{F} \neq \varnothing$
- Generalized Büchi : $\mathcal{F} \subseteq 2^{\mathcal{Q}}$,
  i.e. $\mathcal{F} = \{F_1, \ldots, F_m\}$
  For each F_i, $\underline{\inf}(\rho) \cap F_i \neq \varnothing$
- Rabin : $\mathcal{F} \subseteq 2^{\mathcal{Q}} \times 2^{\mathcal{Q}}$,
  i.e. $\mathcal{F} = \{(G_1, B_1) \ldots, (G_m, B_m)\}$
  For some pair $(G_i, B_i) \in \mathcal{F}$, $\underline{\inf}(\rho) \cap G_i \neq \varnothing \wedge \underline{\inf}(\rho) \cap B_i = \varnothing$
- Streett : $\mathcal{F} \subseteq 2^{\mathcal{Q}} \times 2^{\mathcal{Q}}$,
  i.e. $\mathcal{F} = \{(G_1, B_1) \ldots, (G_m, B_m)\}$
  For all pairs $(G_i, B_i) \in \mathcal{F}$, $\underline{\inf}(\rho) \cap G_i = \varnothing \vee \underline{\inf}(\rho) \cap B_i \neq \varnothing$

ω-regular languages

For nondeterministic automata, all define the ω-regular languages : $\bigcup_i \alpha_i \beta_i^\omega$ where α_i and β_i are finite-word regular languages and ω denotes infinite repetition

Plan

Properties of Büchi automata

Büchi automata are closed under union

- Given $\mathcal{A}_1$ and $\mathcal{A}_2$ with disjoint states set $\mathcal{Q}_1$ and $\mathcal{Q}_2$
- It is easy to define $\mathcal{A}$ with
  - union of states,
  - union of initial states
  - union of accepting states
  - union of transition relation
- $\mathcal{L}(\mathcal{A}) = \mathcal{L}(\mathcal{A}_1) \cup \mathcal{L}(\mathcal{A}_2)$

Properties of Büchi automata

Büchi automata are closed under intersection

- Given the Büchi automata $\mathcal{A}_1 = (\Sigma, \mathcal{Q}_1, \mathcal{I}_1, \mathcal{R}_1, \mathcal{F}_1)$ and $\mathcal{A} = (\Sigma, \mathcal{Q}_2, \mathcal{I}_2, \mathcal{R}_2, \mathcal{F}_2)$
- We can define $\mathcal{A} = (\Sigma, \mathcal{Q}, \mathcal{I}, \mathcal{R}, \mathcal{F})$ with
  - $\mathcal{Q} = \mathcal{Q}_1 \times \mathcal{Q}_2 \times \{1, 2\}$,
  - $\mathcal{I} = \mathcal{I}_1 \times \mathcal{I}_2 \times \{1\}$,
  - $\mathcal{F} = \{\mathcal{F}_1 \times \mathcal{Q}_2 \times \{1\}$
  - $\forall s, s' \in \mathcal{Q}_1, t, t' \in \mathcal{Q}_2, a \in \Sigma, i, j \in \{1, 2\}$: $((s, t, i), a, (s', t', j)) \in \mathcal{R} \iff (s, a, s') \in \mathcal{R}_1, (t, a, t') \in \mathcal{R}_2$, and
    1. $i = 1, s \in \mathcal{F}_1$, and $j = 2$ or
    2. $i = 2, t \in \mathcal{F}_2$, and $j = 2$ or
    3. neither 1 or 2 above applies and $j = i$
- $\mathcal{L}(\mathcal{A}) = \mathcal{L}(\mathcal{A}_1) \cap \mathcal{L}(\mathcal{A}_2)$

Properties of Büchi automata

Büchi automata are closed under complementation

- difficult ! (not seen here)

Nonemtyness for Büchi automata : easy to decide

- Check if some accepting state is accessible from an initial state and nontrivially from itself
- Complexity : linear time

From generalized Büchi to Büchi

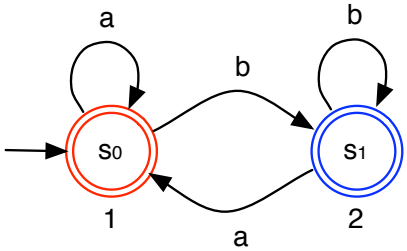
From generalized Büchi to Büchi

Given $\mathcal{A} = (\Sigma, \mathcal{Q}, \mathcal{I}, \mathcal{R}, \mathcal{F})$ where $\mathcal{F} = \{F_1, \ldots, F_k\}$
$\mathcal{A}' = (\Sigma, \mathcal{Q}', \mathcal{I}', \mathcal{R}', \mathcal{F}')$ where

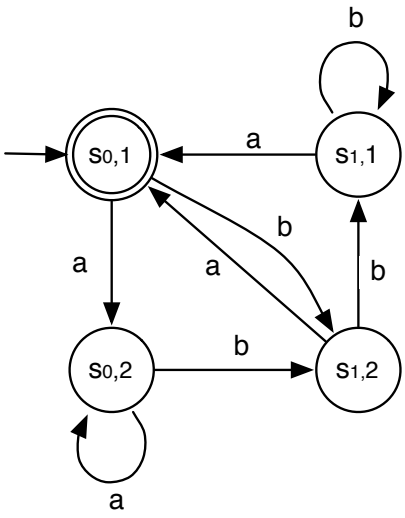
- $\mathcal{Q}' = \mathcal{Q} \times \{1, \ldots, k\}$
- $\mathcal{I}' = \mathcal{I} \times \{1\}$
- $\mathcal{R}'$ is defined by $((s, j), a, (t, i)) \in \mathcal{R}'$ if $(s, a, t) \in \mathcal{R}$ and
  - $i = j$ if $s \notin F_i$
  - $i = (j \bmod k) + 1$ if $s \in F_i$
- $\mathcal{F}' = F_1 \times \{1\}$

$\mathcal{A} \equiv \mathcal{A}'$

From generalized Büchi to Büchi, example



$\mathcal{F} = \{\{s_0\}, \{s_1\}\}$



$\mathcal{F} = \{(s_0, 1)\}$

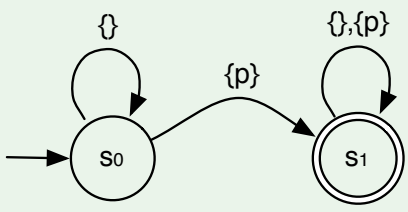
Plan

Büchi automaton of an LTLformula ϕ

Given a (restricted) LTLformula ϕ.
We want to build a Büchi automaton $\mathcal{A}_\phi$ with

$$w \in \mathcal{L}(\mathcal{A}_\phi) \iff w \models \phi$$

Büchi automaton for $\Diamond p$



Construction of the Büchi automaton of an Ltl formula ϕ

Principle on the construction of $\mathcal{A}_\phi$

- A state of $\mathcal{A}_\phi$ is a set of "compatible" subformulae of ϕ
- A transition between two states of $\mathcal{A}_\phi$ is possible when it does respect this "compatibility"
- Initial states of $\mathcal{A}_\phi$ are the one which contains ϕ

Construction of $\mathcal{A}_\phi$

Construction of $\mathcal{A}_\phi$

4 steps :

1. Construction of the local automaton for ϕ
2. Construction of the eventualities automaton for ϕ
3. Composition of both automaton to build a Generalized Büchi automaton
4. Tranformation for the result into a simple Büchi automaton

Construction of the local automaton for $\mathcal{A}_\phi$

Closure of ϕ

- $\phi \in cl(\phi)$
- $\phi_1 \wedge \phi_2 \in cl(\phi) \Rightarrow \phi_1, \phi_2 \in cl(\phi)$
- $\phi_1 \vee \phi_2 \in cl(\phi) \Rightarrow \phi_1, \phi_2 \in cl(\phi)$
- $\bigcirc \phi \in cl(\phi) \Rightarrow \phi \in cl(\phi)$
- $\phi_1 \, \mathrm{U} \, \phi_2 \in cl(\phi) \Rightarrow \phi_1, \phi_2 \in cl(\phi)$
- $\phi_1 \, \tilde{\mathrm{U}} \, \phi_2 \in cl(\phi) \Rightarrow \phi_1, \phi_2 \in cl(\phi)$

Construction of the local automaton for $\mathcal{A}_\phi$

States of $\mathcal{A}_\phi$

The set of states are all "compatible" subset of $cl(\phi)$:

- $\phi_1 \wedge \phi_2 \in s \Rightarrow \phi_1 \in s \wedge \phi_2 \in s$
- $\phi \in s \iff \neg\phi \notin s$
- $\phi_1 \vee \phi_2 \in s \Rightarrow \phi_1 \in s \vee \phi_2 \in s$
- $\phi_1 \, \mathrm{U} \, \phi_2 \in s \Rightarrow \phi_1 \in s \vee \phi_2 \in s$
- $\phi_1 \, \tilde{\mathrm{U}} \, \phi_2 \in s \Rightarrow \phi_2 \in s$

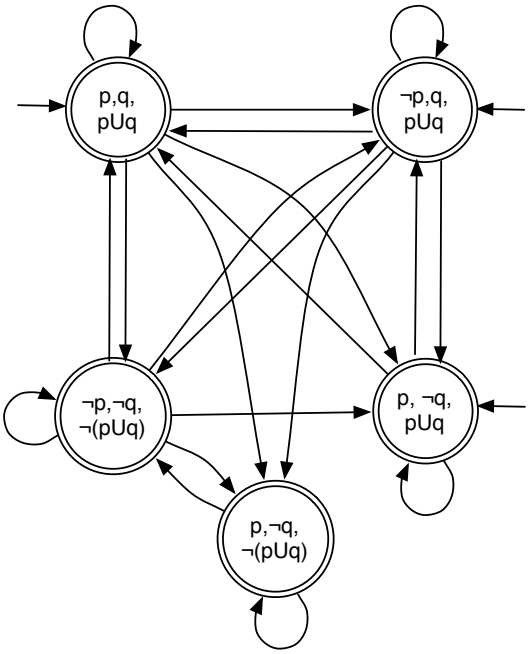
Transition between states of the local automaton for $\mathcal{A}_\phi$

The set of transitions are all "compatible" ones, i.e. if $(s, t) \in \mathcal{R}_\phi$ then

- If $\bigcirc \phi_1 \in s \Rightarrow \phi_1 \in t$
- If $\phi_1 \, \mathrm{U} \, \phi_2 \in s \wedge \phi_2 \notin s \Rightarrow \phi_1 \, \mathrm{U} \, \phi_2 \in t$
- If $\phi_1 \, \tilde{\mathrm{U}} \, \phi_2 \in s \wedge \phi_1 \notin s \Rightarrow \phi_1 \, \tilde{\mathrm{U}} \, \phi_2 \in t$

Construction of local automaton for $\mathcal{A}_\phi$

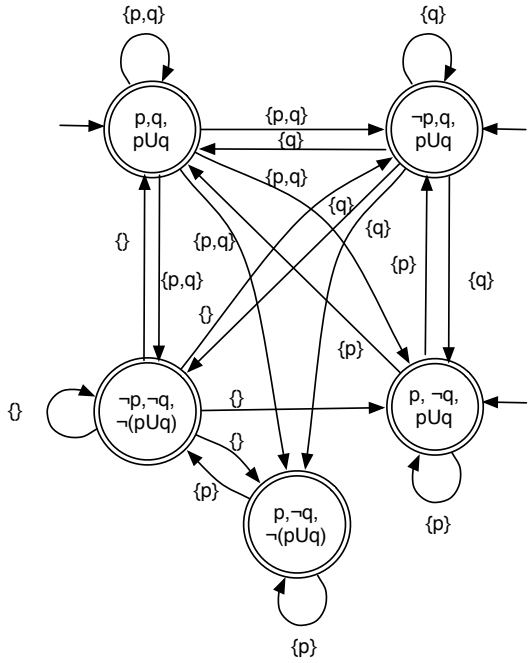
The local automaton for $p \,\mathrm{U}\, q$



local automaton for $p \,\mathrm{U}\, q$ (without label on transitions)

Construction of local automaton for $\mathcal{A}_\phi$

The local automaton for $p \,\mathrm{U}\, q$



local automaton for $p \,\mathrm{U}\, q$

Construction of eventualities automaton for $\mathcal{A}_\phi$

Goal : check that all the $\phi_1 \,\mathrm{U}\, \phi_2$ are "finalized", i.e. for each $\phi_1 \,\mathrm{U}\, \phi_2$ find a state where ϕ_2 is true.

eventualities of ϕ : $ev(\phi)$

- subset of $cl(\phi)$ of the form $\phi_1 \,\mathrm{U}\, \phi_2$

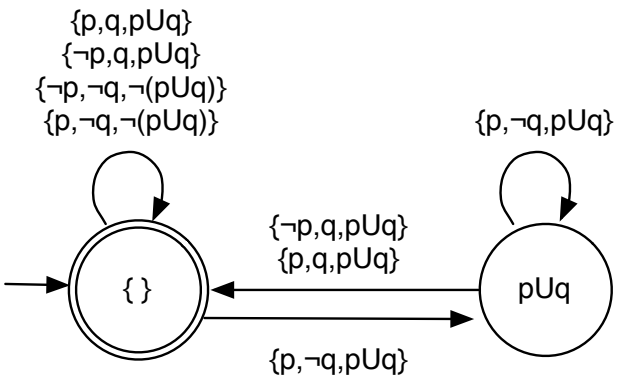
Eventualities automaton for ϕ

$\mathcal{E} = (2^{ev(\phi)}, \mathcal{R}, \{\varnothing\}, \{\varnothing\})$ with
$(s, A, t) \in \mathcal{R}(A \in 2^{cl(\phi)})$ if

- $s = \varnothing \Rightarrow \forall \phi_1 \,\mathrm{U}\, \phi_2 \in ev(\phi) : \phi_1 \,\mathrm{U}\, \phi_2 \in t$ iff $\phi_2 \notin A$
- $s \neq \varnothing \Rightarrow \forall \phi_1 \,\mathrm{U}\, \phi_2 \in s : \phi_1 \,\mathrm{U}\, \phi_2 \in t$ iff $\phi_2 \notin A$

Construction of eventualities automaton for $\mathcal{A}_\phi$

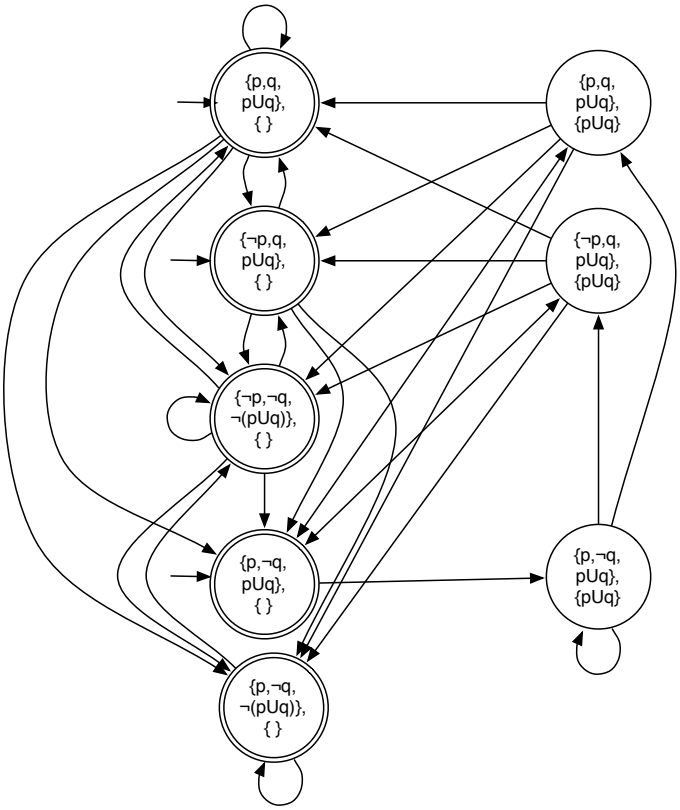
The eventualities automaton for $p \,\mathrm{U}\, q$



eventualities automaton for $p \,\mathrm{U}\, q$

Construction of $\mathcal{A}_\phi$ for $p \,\mathrm{U}\, q$

Since, there is only one until operator in $\phi = p \,\mathrm{U}\, q$, the composition of the local and eventualities automata gives directly a simple Büchi automaton



Chapter 5 : μ-calculus

Plan

Order and partially ordered sets (posets)

Order

- Order : binary relation over $\mathcal{D}$ with the properties of
  - Reflexivity
  - Antisymmetry
  - Transitivity
- Total order : order with $\forall x, y \in \mathcal{D}. x \sqsubseteq y \ \vee \ y \sqsubseteq x$

Partial order set (or poset)

Pair $(\mathcal{D}, \sqsubseteq)$ where $\mathcal{D}$ is a set and $\sqsubseteq$ a binary order relation over $\mathcal{D}$

Example of posets

- $\leq$ on $\mathbb{N}$ is poset (with a total order)
- $(\mathbb{N} \times \mathbb{N}, \sqsubseteq)$ with $(x, y) \sqsubseteq (x', y') \iff x \leq x' \ \wedge \ y \leq y'$
- $(2^S, \subseteq)$ with a set S

(least) upper bound and (greatest) lower bound

Given a poset $(\mathcal{D}, \sqsubseteq)$

Bounds

- $m \in \mathcal{D}$ is an upper bound of $M \subseteq \mathcal{D}$ if $\forall x \in M.x \sqsubseteq m$
- $m \in \mathcal{D}$ is the least upper bound (lub, $sup(M)$, $\sqcup M$) of $M \subseteq \mathcal{D}$ if
  - $\forall x \in M.x \sqsubseteq m$ and
  - $\forall y \in \mathcal{D}.(\forall x \in M.x \sqsubseteq y) \rightarrow m \sqsubseteq y$
- remarks :
  - upper bound may not exists
  - M may have an upper bound, but not a least upper bound
- lower bound and greatest lower bounds (glb, $inf(M)$, $\sqcap M$,) are defined analogously

Properties of bounds

Properties of bounds

- $\sqcup(\bigcup_{i \in I} A_i) = \sqcup(\bigcup_{i \in I} \sqcup A_i)$
- $\sqcap(\bigcup_{i \in I} A_i) = \sqcap(\bigcup_{i \in I} \sqcap A_i)$
- $A \subseteq B$ implies $\sqcup A \sqsubseteq \sqcup B$
- $A \subseteq B$ implies $\sqcap B \sqsubseteq \sqcap A$

(Complete) Lattice

(Complete) Lattice

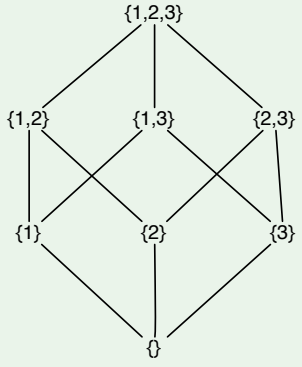
- Given a poset $(\mathcal{D}, \sqsubseteq)$
  - $(\mathcal{D}, \sqsubseteq)$ is a directed set if all $\{x, y\} \subseteq \mathcal{D}$ have a lower and upper bounds in $\mathcal{D}$
  - $(\mathcal{D}, \sqsubseteq)$ is a lattice if all $\{x, y\} \subseteq \mathcal{D}$ have $\sqcup\{x, y\}$ and $\sqcap\{x, y\}$
  - $(\mathcal{D}, \sqsubseteq)$ is a complete lattice if for all non empty $M \subseteq \mathcal{D}$ have $\sqcup M$ and $\sqcap M$
- In a lattice, for every finite set M_{fin}
  - $\sqcup(\{e\} \cup M_{fin}) = \sqcup\{e, \sqcup M_{fin}\}$
  - $\sqcap(\{e\} \cup M_{fin}) = \sqcap\{e, \sqcap M_{fin}\}$
- In a lattice, $\sqcup M_{fin}$ and $\sqcap M_{fin}$ exist for finite M_{fin}
- In a complete lattice, we define $\bot := \sqcap\mathcal{D}$ and $\top := \sqcup\mathcal{D}$

Example of lattice and complete lattice

Example of lattice and comlete lattice

- Every total order is a lattice
- $(\mathbb{N}, \leq)$ is a lattice
- $(\mathbb{N} \cup \{\top\}, \sqsubseteq)$ with $n \sqsubseteq m \iff n \leq m \vee m = \top$ is a complete lattice
- $(2^S, \subseteq)$ with a finite set S is a complete lattice

Example : with $S = \{1, 2, 3\}$



Algebraic properties of Lattices

Given x, y

- Commutativity : $x \sqcap y = y \sqcap x$ $(x \sqcup y = y \sqcup x)$
- Associativity : $x \sqcap (y \sqcap z) = (x \sqcap y) \sqcap z$ $(x \sqcup (y \sqcup z) = (x \sqcup y) \sqcup z)$
- Absorption : $x \sqcap (x \sqcup y) = x$ $(x \sqcup (x \sqcap y) = x)$
- Idempotency : $x \sqcap x = x$ $(x \sqcup x = x)$

Important lattice : The set of subset of a set S

In the lattice $(2^S, \subseteq)$

- $S_1 \sqcup S_2 = S_1 \cup S_2$
- $S_1 \sqcap S_2 = S_1 \cap S_2$
- $\bot := \{\}$ and $\top := S$

Monotonic and Continuous Functions and fixpoint

Monotonic and Continuous Functions and fixpoint

- Given complete lattices $(\mathcal{D}, \sqsubseteq_{\mathcal{D}})$ and $(\mathcal{E}, \sqsubseteq_{\mathcal{E}})$
- given a function $f : \mathcal{D} \to \mathcal{E}$
- f is monotonic if $x \sqsubseteq_{\mathcal{D}} y \to f(x) \sqsubseteq_{\mathcal{E}} f(y)$
- f is continuous if $f(\sqcup M) = \sqcup f(M)$ and $f(\sqcap M) = \sqcap f(M)$ hold for every directed set $M \neq \{\}$
- $x \in \mathcal{D}$ is a fixpoint of $f : \mathcal{D} \to \mathcal{D}$ if $f(x) = x$ holds

Properties (with $\mathcal{D}$ a complete lattice)

- Every continuous function is monotonic
- If $\mathcal{D}$ is finite, every monotonic function is continuous
- Every monotonic function $f : \mathcal{D} \to \mathcal{D}$ has fixpoints

Computation of Fixpoints

Computation of Fixpoints (Tarski-Knaster Theorem)

- Given $\mathcal{D}$ a complete lattice and continuous $f : \mathcal{D} \to \mathcal{D}$
- We write $\mu x.f(x)$ and $\nu x.f(x)$ the least resp. greatest fixpoint of f
- The sequence $p_{i+1} := f(p_i)$ with $p_0 := \bot$ converges to $\mu x.f(x)$
- The sequence $q_{i+1} := f(q_i)$ with $q_0 := \top$ converges to $\nu x.f(x)$
- $\mu x.f(x) = \sqcap(\{x \in \mathcal{D} \mid f(x) \sqsubseteq x\})$
- $\nu x.f(x) = \sqcup(\{x \in \mathcal{D} \mid f(x) \sqsubseteq x\})$

Computation of fixpoints with finite complete lattice

Given a finite lattice $(\mathcal{D}, \sqsubseteq)$ and a continuous function f

- $\mu x.f(x) = f^m(\bot)$ for some natural number m
- $\nu x.f(x) = f^M(\top)$ for some natural number M

Plan

What is the μ-calculus ?

What is the μ-calculus ?

- Class of temporal logics
- Used to describe and verify properties of Kripke structures or Labeled Transitions Systems
- Uses fixpoint operators
- Many temporal logics can be translated into μ-calculus (e.g. LTL, CTL, CTL*)

Syntax of the μ-calculus

Note : the following definition is a possible μ-calculus ; other operators, could be defined

Set of μ-calculus formulae $\mathcal{L}_\mu$

Given variables $\mathcal{V}, x \in \mathcal{V}, \phi, \psi \in \mathcal{L}_\mu$

$$\top \mid \bot \mid \quad x \mid \neg\phi \mid \phi \wedge \psi \mid \phi \vee \psi \mid \langle\rangle\phi \mid [\,]\phi \mid \mu x.\phi \mid \nu x.\phi$$

Semantics of the μ-calculus

Semantics I

Given

- the Kripke structure $\mathcal{K} = (\mathcal{I}, \mathcal{S}, \mathcal{R}, \mathcal{L})$ on variables $\mathcal{V}$
- a fixpoint-free formula $\phi \in L_\mu$ over variables $\mathcal{V}$

$[\![\phi]\!]_\mathcal{K}$ gives the set of states which satisfies ϕ

- $[\![\top]\!]_\mathcal{K} := \mathcal{S}$
- $[\![\bot]\!]_\mathcal{K} := \varnothing$
- $[\![x]\!]_\mathcal{K} := \{s \in \mathcal{S} \mid x \in \mathcal{L}(s)\}$
- $[\![\neg\phi]\!]_\mathcal{K} := \mathcal{S} \setminus [\![\phi]\!]_\mathcal{K}$
- $[\![\phi \wedge \psi]\!]_\mathcal{K} := [\![\phi]\!]_\mathcal{K} \cap [\![\psi]\!]_\mathcal{K}$
- $[\![\phi \vee \psi]\!]_\mathcal{K} := [\![\phi]\!]_\mathcal{K} \cup [\![\psi]\!]_\mathcal{K}$
- $[\![\langle\rangle\phi]\!]_\mathcal{K} := \mathrm{pre}^\mathcal{R}_\exists([\![\phi]\!]_\mathcal{K})$ (set of states which have a succesor in $[\![\phi]\!]$)
- $[\![[\,]\phi]\!]_\mathcal{K} := \mathrm{pre}^\mathcal{R}_\forall([\![\phi]\!]_\mathcal{K})$ (set of states which have all succesors in $[\![\phi]\!]$)

Semantics of the μ-calculus

Modified structure $\mathcal{K}_x^Q$

Given the Kripke structure $\mathcal{K} = (\mathcal{I}, \mathcal{S}, \mathcal{R}, \mathcal{L})$ and

a set of states $Q \subseteq \mathcal{S}$,

intuitively, $\mathcal{K}_x^Q$ corresponds to the Kripke structure $\mathcal{K}$ where we have "added" a proposition x which is true in states Q.

To simplify we suppose x is not used as a simple proposition

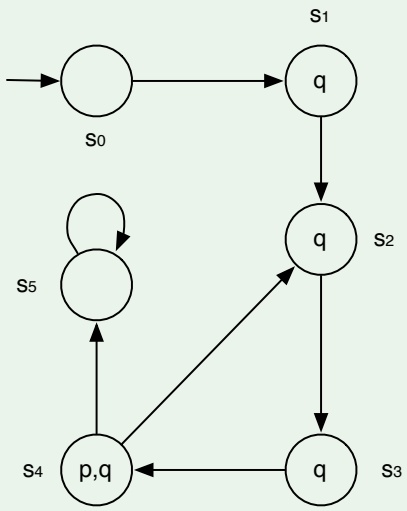
$\mathcal{K}_x^Q := (\mathcal{I}, \mathcal{S}, \mathcal{R}, \mathcal{L}_x^Q)$ with

$$\mathcal{L}_x^Q(s) := \begin{cases} \mathcal{L}(s) & : \text{ if } s \notin Q \\ \mathcal{L}(s) \cup \{x\} & : \text{ if } s \in Q \end{cases}$$

- With this definition we have $[\![x]\!]_{\mathcal{K}_x^Q} := Q$
- $f(Q) := [\![\phi]\!]_{\mathcal{K}_x^Q}$ is a state transformer (maps each set of states to a set of states)

Examples :

Example of $[\![\phi]\!]_{\mathcal{K}_x^Q}$



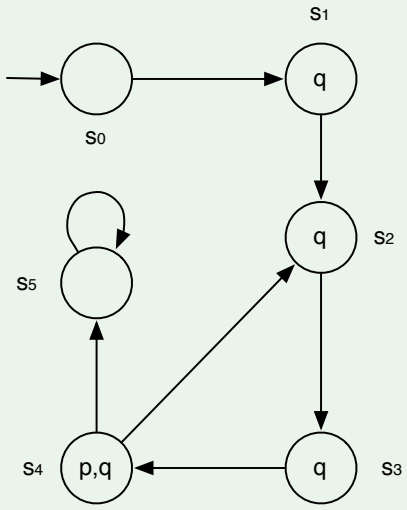
With $\mathcal{K}$ and $\phi = \langle\rangle(x \vee p)$

Q	$[\![\phi]\!]_{\mathcal{K}_x^Q}$
$\varnothing$	$\{s_3\}$
$\{s_3\}$	$\{s_2, s_3\}$
$\{s_2, s_3\}$	$\{s_1, s_2, s_3, s_4\}$
$\{s_1, s_2, s_3, s_4\}$	$\{s_0, s_1, s_2, s_3, s_4\}$
$\{s_0, s_1, s_2, s_3, s_4\}$	$\{s_0, s_1, s_2, s_3, s_4\}$

For $Q = \{s_0, s_1, s_2, s_3, s_4\}$,
$Q = [\![\phi]\!]_{\mathcal{K}_x^Q}$ (Q is a fixpoint for $[\![\phi]\!]_{\mathcal{K}_x^Q}$)

Examples :

Example of $[\![\phi]\!]^Q_{\mathcal{K}_x}$



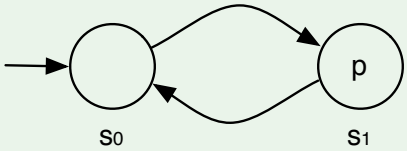
With $\mathcal{K}$ and $\phi = [\,](x \wedge q)$

Q	$[\![\phi]\!]^Q_{\mathcal{K}_x}$
S	$\{s_0, s_1, s_2, s_3\}$
$\{s_0, s_1, s_2, s_3\}$	$\{s_0, s_1, s_2\}$
$\{s_0, s_1, s_2\}$	$\{s_0, s_1\}$
$\{s_0, s_1\}$	$\{s_0\}$
$\{s_0\}$	$\varnothing$
$\varnothing$	$\varnothing$

For $Q = \varnothing$,
$Q = [\![\phi]\!]^Q_{\mathcal{K}_x}$ (Q is a fixpoint for $[\![\phi]\!]^Q_{\mathcal{K}_x}$)

Examples :

Example of $[\![\phi]\!]^Q_{\mathcal{K}_x}$



With $\mathcal{K}$ and $\phi = \neg x$

Q	$[\![\phi]\!]^Q_{\mathcal{K}_x}$
$\varnothing$	S
S	$\varnothing$
$\{s_0\}$	$\{s_1\}$
$\{s_1\}$	$\{s_0\}$

No fixpoint here !

Semantics of the μ-calculus

Semantics II

Given

- $[\![\top]\!]_{\mathcal{K}} := \mathcal{S}$
- $[\![\bot]\!]_{\mathcal{K}} := \varnothing$
- $[\![x]\!]_{\mathcal{K}} := \{s \in \mathcal{S} \mid x \in \mathcal{L}(s)\}$
- $[\![\neg\phi]\!]_{\mathcal{K}} := \mathcal{S} \setminus [\![\phi]\!]_{\mathcal{K}}$
- $[\![\phi \wedge \psi]\!]_{\mathcal{K}} := [\![\phi]\!]_{\mathcal{K}} \cap [\![\psi]\!]_{\mathcal{K}}$
- $[\![\phi \vee \psi]\!]_{\mathcal{K}} := [\![\phi]\!]_{\mathcal{K}} \cup [\![\psi]\!]_{\mathcal{K}}$
- $[\![\langle\rangle\phi]\!]_{\mathcal{K}} := \mathrm{pre}^{\mathcal{R}}_{\exists}([\![\phi]\!]_{\mathcal{K}})$ (set of states which have a succesor in $[\![\phi]\!]$)
- $[\![[]\phi]\!]_{\mathcal{K}} := \mathrm{pre}^{\mathcal{R}}_{\forall}([\![\phi]\!]_{\mathcal{K}})$ (set of states which have all succesors in $[\![\phi]\!]$)
- $[\![\mu x.\phi]\!]_{\mathcal{K}}$ is the least fixpoint of $f(Q) := [\![\phi]\!]_{\mathcal{K}^Q_x}$
- $[\![\nu x.\phi]\!]_{\mathcal{K}}$ is the greatest fixpoint of $f(Q) := [\![\phi]\!]_{\mathcal{K}^Q_x}$

Existence of fixpoints

Existence of fixpoints

- Not every function has fixpoints
- We have seen that not every state transformer has fixpoint
- Since $\mathcal{K}$ has a finite set of state, a sufficient condition to have fixpoint is $f(Q) := [\![\phi]\!]_{\mathcal{K}^Q_x}$ is monotonic
- It is the case if x has only positive occurrences of ϕ
  i.e. x is always nested in an even number of negations.

Example of μ-calculus formulae

Properties specified by μ-calculus formulae

- Invariance (ϕ always true) : $\nu x.(\phi \wedge [\,]x)$
- Reachability (ϕ reachable) : $\mu x.(\phi \vee \langle\,\rangle x)$
- Persistence (ϕ is reachable and then remains always true) : $\mu y.[\nu x.(\phi \wedge [\,]x) \vee \langle\,\rangle y]$

μ-calculus model checking

Principle

- Compute the set of states in $\mathcal{K}$, which satisfy subformulae ϕ_i of ϕ
- $\mathcal{K} \models \phi \iff \mathcal{I} \subseteq [\![\phi]\!]_{\mathcal{K}}$
- see chapter 6.

Chapter 6 : Model Checking

Plan

From previous chapter

LTL syntax

Given a set of propositions $\mathcal{P}$, a formula in Linear Temporal Logic (LTL) is defined using the following grammar :

$$\phi ::= \top \mid \bot \mid \quad p \mid \neg\phi \mid \phi \vee \phi \mid \phi \wedge \phi \mid \bigcirc \phi \mid \phi \,\mathrm{U}\, \phi \mid \phi \,\tilde{\mathrm{U}}\, \phi$$

where $p \in \mathcal{P}$.

Principle of LTL model checking

- $S \models \phi \equiv [\![S]\!] \subseteq [\![\phi]\!] \equiv [\![S]\!] \cap [\![\neg\phi]\!] = \varnothing$
- Since $[\![S]\!]$ and $[\![\phi]\!]$ are infinite sets, the idea is to work with automata
- S is a Kripke structure ✓
- For $\neg\phi$? : Büchi automata (see chapter 4)
- Check that $\mathcal{L}(S \times \mathcal{B}_{\neg\phi}) = \varnothing$

LTL model checking

Complexity of the LTL model checking

- The size of the Büchi automaton for $\neg\phi$ is (in the worst case), in $\mathcal{O}(2^{|\phi|})$
- The size of $S \times \mathcal{B}_{\neg\phi}$ is in $\mathcal{O}(|S|.|\mathcal{B}\neg\phi|)$
- Cheching if $\mathcal{L}(S \times \mathcal{B}_{\neg\phi}) = \varnothing$ is linear in the size of $S \times \mathcal{B}_{\neg\phi}$
- The resulting complexity is in $\mathcal{O}(|S|.2^{|\phi|})$ (linear in the size of the system, exponential in the size of the formula)

Plan

From previous chapter

CTL syntax

Given a set of propositions $\mathcal{P}$, a formula in Computation Tree Logic (CTL) is defined using the following grammar :

$$\phi ::= \top \mid p \mid \neg\phi \mid \phi \vee \phi \mid \exists \bigcirc \phi \mid \forall \bigcirc \phi \mid \exists \phi \mathbf{U} \phi \mid \forall \phi \mathbf{U} \phi$$

where $p \in \mathcal{P}$.

Principle of CTL model checking

- For all state s in $\mathcal{K}$, decorate s with all the subformulae ϕ_i of ϕ such that $s \models \phi_i$

CTL without universal quantifiers

CTL without universal quantifiers (is sometimes useful)

In every CTL formula, every universal quantifiers can be replaced using the following equivalence :

- $\forall \bigcirc \phi \equiv \neg\exists \bigcirc \neg\phi$
- $\forall\phi_1 \,\mathrm{U}\, \phi_2 \equiv \neg\exists(\neg\phi_2 \,\mathrm{U}\, (\neg\phi_1 \wedge \neg\phi_2)) \wedge \neg\exists\Box\neg\phi_2)$
- $\forall\Box\phi \equiv \neg\exists\Diamond\neg\phi$
- $\forall\Diamond\phi \equiv \neg\exists\Box\neg\phi$

CTL model checking

Idea : extend the labeling $\mathcal{L}(s)$ with all subformulae ϕ_i of ϕ such that $s \models \phi_i$

Structure of the algorithm

Inductive :

- basis :
  - $\top$ is true in all states ;
  - $\bot$ is false in all states ;
  - $\mathcal{L}(s)$ gives the sets of propositions true in s ;
- Induction : 6 cases :
  - $\neg\phi \in \mathcal{L}(s)$ iff $\phi \notin \mathcal{L}(s)$
  - $\phi_1 \vee \phi_2 \in \mathcal{L}(s)$ iff $\phi_1 \in \mathcal{L}(s) \vee \phi_2 \in \mathcal{L}(s)$
  - $\exists \bigcirc \phi \in \mathcal{L}(s)$ iff $\exists t \in \mathrm{suc}^{\mathcal{R}}_{\exists}(s).\phi \in \mathcal{L}(t)$
  - $\forall \bigcirc \phi \in \mathcal{L}(s)$ iff $\forall t \in \mathrm{suc}^{\mathcal{R}}_{\forall}(s).\phi \in \mathcal{L}(t)$
  - $\exists\phi_1 \,\mathrm{U}\, \phi_2$: see algorithm below
  - $\forall\phi_1 \,\mathrm{U}\, \phi_2$: see algorithm below

Algorithm for $s \models \forall \phi_1 \, \mathrm{U} \, \phi_2$

Require: $\forall t \in S.\neg \mathit{marked}(t) \wedge \forall i \in \{1,2\}.(\phi_i \in \mathcal{L}(t) \iff t \models \phi_i)$
Ensure: return true $\wedge (\forall \phi_1 \, \mathrm{U} \, \phi_2 \in \mathcal{L}(s) \iff s \models \forall \phi_1 \, \mathrm{U} \, \phi_2)$
  if $(\forall \phi_1 \, \mathrm{U} \, \phi_2) \in \mathcal{L}(s)$ **then**
    return true
  else if $\neg \mathit{marked}(s)$ **then**
    if $\phi_2 \in \mathcal{L}(s)$ **then**
      $\mathcal{L}(s) \Leftarrow (\forall \phi_1 \, \mathrm{U} \, \phi_2)$
      return true
    else if $\phi_1 \notin \mathcal{L}(s)$ **then**
      return false
    else
      $\mathit{marked}(s) \leftarrow$ **true**
      if $\forall t \in \mathrm{suc}^{\mathcal{R}}_{\exists}(s).t \models \forall \phi_1 \, \mathrm{U} \, \phi_2$ **then**
        $\mathcal{L}(s) \Leftarrow (\forall \phi_1 \, \mathrm{U} \, \phi_2)$
        return true
      else
        return false
      end if
    end if
  else
    return false
  end if

Algorithm for $s \models \exists \phi_1 \, \mathrm{U} \, \phi_2$

Require: $\forall t \in S.\neg \mathit{marked}(t) \wedge \forall i \in \{1,2\}.(\phi_i \in \mathcal{L}(t) \iff t \models \phi_i)$
Ensure: return true $\wedge (\exists \phi_1 \, \mathrm{U} \, \phi_2 \in \mathcal{L}(s) \iff s \models \exists \phi_1 \, \mathrm{U} \, \phi_2)$
  if $(\forall \phi_1 \, \mathrm{U} \, \phi_2) \in \mathcal{L}(s)$ **then**
    return true
  else if $\neg \mathit{marked}(s)$ **then**
    if $\phi_2 \in \mathcal{L}(s)$ **then**
      $\mathcal{L}(s) \Leftarrow (\exists \phi_1 \, \mathrm{U} \, \phi_2)$
      return true
    else if $\phi_1 \notin \mathcal{L}(s)$ **then**
      return false
    else
      $\mathit{marked}(s) \leftarrow$ **true**
      if $\exists t \in \mathrm{suc}^{\mathcal{R}}_{\exists}(s).t \models \exists \phi_1 \, \mathrm{U} \, \phi_2$ **then**
        $\mathcal{L}(s) \Leftarrow (\exists \phi_1 \, \mathrm{U} \, \phi_2)$
        return true
      else
        return false
      end if
    end if
  else
    return false
  end if

Ctl Model checking

More efficient method

A more efficient symbolic method is defined through a CTL to μ-calculus translation (see below)

Plan

From previous chapter

μ-calculus syntax

Given variables $\mathcal{V}, x \in \mathcal{V}, \phi, \psi \in \mathcal{L}_\mu$

$$\top \mid \bot \mid \quad p \mid x \mid \neg\phi \mid \phi \wedge \psi \mid \phi \vee \psi \mid \langle\rangle\phi \mid []\phi \mid \mu x.\psi \mid \nu x.\psi$$

Principle of the μ-calculus model checking

- Compute the set of states in $\mathcal{K}$, which satisfy subformulae ϕ_i of ϕ
- $\mathcal{K} \models \phi \iff \mathcal{I} \subseteq [\![\phi]\!]_\mathcal{K}$

Algorithm which computes $States_\mu(\phi)$

Case $\phi \equiv$

$\top$	:	**return** $\mathcal{S}$
$\bot$	:	**return** $\varnothing$
x	:	**return** $\{s \in \mathcal{S} \mid x \in \mathcal{L}(s)\}$
$\neg\phi_1$	:	**return** $\mathcal{S} \setminus States_\mu(\phi_1)$
$\phi_1 \wedge \phi_2$	:	**return** $States_\mu(\phi_1) \cap States_\mu(\phi_2)$
$\phi_1 \vee \phi_2$	:	**return** $States_\mu(\phi_1) \cup States_\mu(\phi_2)$
$\langle\rangle\phi_1$	:	**return** $\mathrm{pre}^{\mathcal{R}}_{\exists}(States_\mu(\phi_1))$
$[]\phi_1$	:	**return** $\mathrm{pre}^{\mathcal{R}}_{\forall}(States_\mu(\phi_1))$

$\mu x.\psi$:

$Q_1 := \{\}$;
repeat
  $Q_0 := Q_1$;
  $\mathcal{L} := \mathcal{L}_x^{Q_1}$;
  $Q_1 := States_\mu(\psi)$;
until $Q_0 = Q_1$;
return Q_0;

$\nu x.\psi$:

$Q_1 := \mathcal{S}$;
repeat
  $Q_0 := Q_1$;
  $\mathcal{L} := \mathcal{L}_x^{Q_1}$;
  $Q_1 := States_\mu(\psi)$;
until $Q_0 = Q_1$;
return Q_0;

CTL model checking through a translation into the μ-calculus

CTL to μ-calculus (Clarke and Emerson 1981)

- $\exists \bigcirc \phi = \langle\rangle\phi$
- $\forall \bigcirc \phi = []\phi$
- $\exists\phi_1 \,\mathrm{U}\, \phi_2 = \phi_2 \vee (\phi_1 \wedge \langle\rangle(\exists\phi_1 \,\mathrm{U}\, \phi_2)) = \mu Z.[\phi_2 \vee (\phi_1 \wedge \langle\rangle Z)]$
- $\forall\phi_1 \,\mathrm{U}\, \phi_2 = \phi_2 \vee (\phi_1 \wedge [](\forall\phi_1 \,\mathrm{U}\, \phi_2)) = \mu Z.[\phi_2 \vee (\phi_1 \wedge [] Z)]$
- $\exists\Diamond\phi_1 = \phi_1 \vee \langle\rangle\exists\Diamond\phi_1)) = \mu Z.[\phi_1 \vee \langle\rangle Z]$
- $\forall\Diamond\phi_1 = \phi_1 \vee []\forall\Diamond\phi_1)) = \mu Z.[\phi_1 \vee [] Z]$
- $\exists\Box\phi_1 = \phi_1 \wedge \langle\rangle\exists\Box\phi_1 = \nu Z.[\phi_1 \wedge \langle\rangle Z]$
- $\forall\Box\phi_1 = \phi_1 \wedge []\forall\Box\phi_1 = \nu Z.[\phi_1 \wedge [] Z]$

CTL model checking through a translation into the μ-calculus

CTL to μ-calculus with no use of the [ ] operator

Through the replacement of every universal quantifiers :

- $\forall \bigcirc \phi \equiv \neg\exists \bigcirc \neg\phi$
- $\forall\phi_1 \,\mathrm{U}\, \phi_2 \equiv \neg\exists(\neg\phi_2 \,\mathrm{U}\, (\neg\phi_1 \wedge \neg\phi_2)) \wedge \neg\exists\Box\neg\phi_2)$
- $\forall\Box\phi \equiv \neg\exists\Diamond\neg\phi$
- $\forall\Diamond\phi \equiv \neg\exists\Box\neg\phi$

CTL to μ-calculus (Clarke and Emerson 1981)

- $\exists \bigcirc \phi = \langle\rangle\phi$
- ~~$\forall \bigcirc \phi = []\phi$~~
- $\exists\phi_1 \,\mathrm{U}\, \phi_2 = \phi_2 \vee (\phi_1 \wedge \langle\rangle(\exists\phi_1 \,\mathrm{U}\, \phi_2)) = \mu Z.[\phi_2 \vee (\phi_1 \wedge \langle\rangle Z)]$
- ~~$\forall\phi_1 \,\mathrm{U}\, \phi_2 = \phi_2 \vee (\phi_1 \wedge [](\forall\phi_1 \,\mathrm{U}\, \phi_2)) = \mu Z.[\phi_2 \vee (\phi_1 \wedge [] Z)]$~~
- $\exists\Diamond\phi_1 = \phi_1 \vee \langle\rangle\exists\Diamond\phi_1)) = \mu Z.[\phi_1 \vee \langle\rangle Z]$
- ~~$\forall\Diamond\phi_1 = \phi_1 \vee []\forall\Diamond\phi_1)) = \mu Z.[\phi_1 \vee [] Z]$~~
- $\exists\Box\phi_1 = \phi_1 \wedge \langle\rangle\exists\Box\phi_1 = \nu Z.[\phi_1 \wedge \langle\rangle Z]$
- ~~$\forall\Box\phi_1 = \phi_1 \wedge []\forall\Box\phi_1 = \nu Z.[\phi_1 \wedge [] Z]$~~

Chapter 7 : Symbolic and efficient Model Checking

Plan

Binary encoding of set of states

Binary encoding of set of states

- Global state = { value of each variable } (including the current program execution point)
- Suppose : only static global variables, each variable has a fixed number of bits,

$\Rightarrow$ A state defined by a conjunction which gives the value of each boolean variable $v = (v_1, v_2, \ldots, v_n)$ (e.g. with $n = 5$: $v_1 \wedge \neg v_2 \wedge v_3 \wedge v_4 \wedge \neg v_5$

- A set of states p defined as a predicate.

$$\bigvee_{i \in \{1..|v|\}} \bigwedge_{1 \leq j \leq n} \ell_{ij}$$

with ℓ_{ij} = either v_j or $\neg v_j$
E.g.
$(v_1 \wedge \neg v_2 \wedge v_3 \wedge \neg v_4 \wedge v_5) \vee (\neg v_1 \wedge v_2 \wedge \neg v_3 \wedge v_4 \wedge v_5)$

- Suppose **p** an encoding of this formula for p
We note : $p = \lambda(v)\mathbf{p} = \lambda(v_1, v_2, v_3, v_4, v_5)\mathbf{p}$

Binary encoding of a transition relation

Binary encoding of a transition relation

- A transition = $p \times p'$
- A transition defined by a conjunction of $2n$ litterals : e.g.

$$(v_1 \wedge \neg v_2 \wedge v_3 \wedge \neg v_4 \wedge v_5) \wedge (\neg v'_1 \wedge v'_2 \wedge \neg v'_3 \wedge v'_4 \wedge v'_5)$$

- A transition relation : set of transitions encoded as a predicate.

$$\bigvee_{i \in \{1..|R|\}} \bigwedge_{1 \leq j \leq n} \ell_{ij} \wedge \ell'_{ij}$$

with ℓ_{ij} = either v_j or $\neg v_j$ ℓ'_{ij} = either v'_j or $\neg v'_j$
E.g.
$(v_1 \wedge \neg v_2 \wedge v_3 \wedge \neg v_4 \wedge v_5) \wedge (\neg v'_1 \wedge v'_2 \wedge \neg v'_3 \wedge v'_4 \wedge v'_5) \vee$
$(v_1 \wedge v_2 \wedge \neg v_3 \wedge \neg v_4 \wedge v_5) \wedge (v'_1 \wedge v'_2 \wedge \neg v'_3 \wedge \neg v'_4 \wedge v'_5)$

- Suppose **R :** an encoding of $\mathcal{R}$
We note $R = \lambda(v, v')\mathbf{R} : = \lambda(v_1, v_2, v_3, v_4, v_5, v'_1, v'_2, v'_3, v'_4, v'_5)\mathbf{R}$

Computing $[\![\langle\,\rangle\phi]\!]$ through its encoding

Computing $\langle\,\rangle\phi$

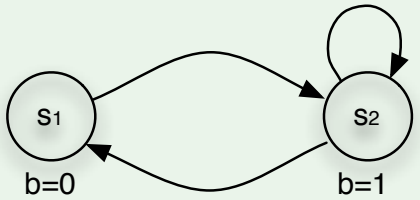
- Given $p(v)$ the binary predicate of $[\![\phi]\!]$
- and $\lambda(v, v')R$ the binary predicate of the transition relation $\mathcal{R}$ of the system $\mathcal{K}$
- The binary predicate of $[\![\langle\,\rangle\phi]\!] = \lambda(v)\exists v'(R(v, v') \wedge p(v'))$

- Given **p** an encoding of $[\![\phi]\!]$ (the set of global states which satisfy ϕ in $\mathcal{K}$;
- and **R** an encoding of the transition relation $\mathcal{R}$ of the system $\mathcal{K}$
- **p'** = **p**$[v_i \leftarrow v_i']$
- The encoding of $[\![\langle\,\rangle\phi]\!] = \lambda v.\exists v'(\mathbf{R} \wedge \mathbf{p'})$

Examples

Example 1 : computing the predicate for $[\![\langle\,\rangle\neg b]\!]$ in $\mathcal{K}$

- $s_1 = \neg b$
- $s_2 = b$
- $R = ((\neg b \wedge b') \vee (b \wedge \neg b') \vee (b \wedge b')) = (b \vee b')$

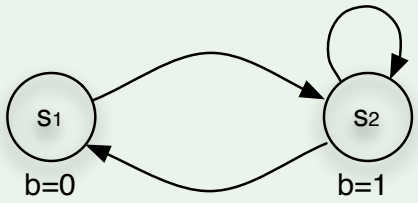


- $\langle\,\rangle\neg b$
  $= \exists b'((b \vee b') \wedge ((\neg b)[b \leftarrow b']))$
  $= \exists b'((b \vee b') \wedge \neg b')$
  $= \exists b'(b \wedge \neg b')$
  $= (b \wedge \neg 0) \vee (b \wedge \neg 1)$
  $= b$ (state s_2)

Examples

Example 2 : computing the predicate for $[\![\exists\Diamond b]\!] = [\![\mu y.(b \vee \langle\rangle y)]\!]$ in $\mathcal{K}$

- $s_1 = \neg b$
- $s_2 = b$
- $R = ((\neg b \wedge b') \vee (b \wedge \neg b') \vee (b \wedge b')) = (b \vee b')$



$f(0) = b \vee \langle\rangle 0 = b$

$f^2(0) = b \vee \langle\rangle b =$
$b \vee \exists b'.((b \vee b') \wedge b')$
$= b \vee (b \vee 1)$
$= 1$ (all states)

$f^3(0) = b \vee \langle\rangle 1 = 1$

CTL model checking revisited

$[\phi]$: predicate of a CTL formula ϕ

Given $[s]$ the predicate for $s \in \mathcal{S}$

$\top$	$\bigvee_{s \in \mathcal{S}} [s]$
p	$\bigvee_{s \in \mathcal{S} \mid p \in \mathcal{L}(s)} [s]$
$\neg\phi$	$\neg[\phi]$
$\phi_1 \vee \phi_2$	$[\phi_1] \vee [\phi_2]$
$\exists \bigcirc \phi_1$	$\mathit{EvalEX}([\phi_1])$
$\exists \phi_1 \, \mathrm{U} \, \phi_2$	$\mathit{EvalEU}([\phi_1], [\phi_2])$
$\exists\Diamond\phi_1$	$\mathit{EvalEF}([\phi_1])$
$\exists\Box\phi_1$	$\mathit{EvalEG}([\phi_1])$

where $[s]$ is the predicate corresponding to state s.

$[\phi]$: predicate of a CTL formula ϕ (cont'd)

$EvalEX(\mathbf{p}) := \exists v'(\mathbf{R} \wedge \mathbf{p'})$

$EvalEU(\mathbf{p}, \mathbf{q}) :=$
$\mathbf{y} = \varnothing$
$\mathbf{y'} = \mathbf{q} \vee (\mathbf{p} \wedge EvalEX(\mathbf{y})$
while$(\mathbf{y} \neq \mathbf{y'})$
$\mathbf{y} = \mathbf{y'}$
$\mathbf{y'} = \mathbf{q} \vee (\mathbf{p} \wedge EvalEX(\mathbf{y})$
return $\mathbf{y}$

$EvalEF(\mathbf{p}) :=$
$\mathbf{y} = \varnothing$
$\mathbf{y'} = \mathbf{p} \vee EvalEX(\mathbf{y})$
while$(\mathbf{y} \neq \mathbf{y'})$
$\mathbf{y} = \mathbf{y'}$
$\mathbf{y'} = \mathbf{p} \vee EvalEX(\mathbf{y})$
return $\mathbf{y}$

$EvalEG(\mathbf{p}) :=$
$\mathbf{y} = [\top]$
$\mathbf{y'} = \mathbf{p} \wedge EvalEX(\mathbf{y})$
while$(\mathbf{y} \neq \mathbf{y'})$
$\mathbf{y} = \mathbf{y'}$
$\mathbf{y'} = \mathbf{p} \wedge EvalEX(\mathbf{y})$
return $\mathbf{y}$

Binary Decision Diagram

Note

Slides done with the help of a tutorial from Henrik Reif Andersen (see web)

Motivation

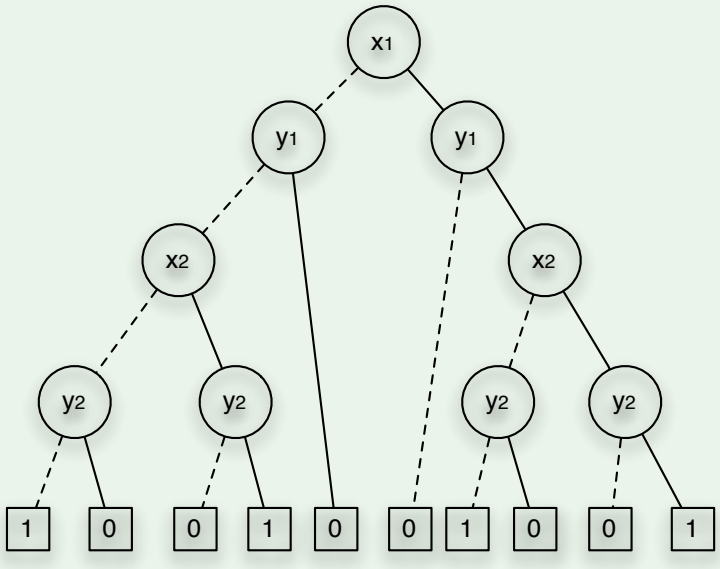
- data structure which gives a compact and efficient encoding of proposition boolean formulae
- The logic operations can directly be done on them

Known results

- Cook's Theorem : Satisfiability of Boolean expressions is NP-complete
- Shannon expansion : given a boolean expression t with a variable x : $t = x \rightarrow t[1/x], t[0/x]$ (with $x \rightarrow y_0, y_1 = (x \wedge y_0) \vee (\neg x \wedge y_1)$

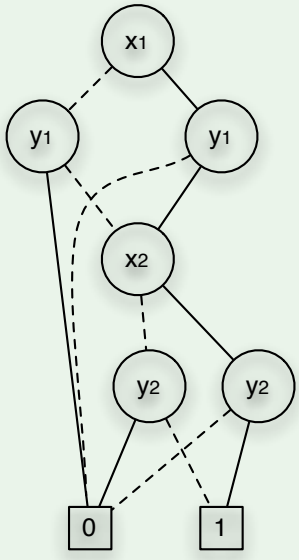
Formula as decision tree

Decision tree of $(x_1 \leftrightarrow y_1) \wedge (x_2 \leftrightarrow y_2)$ with the order in the variables $x_1 < y_1 < x_2 < y_2$



Formula as BDD

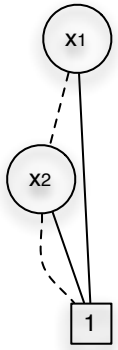
BDD of $(x_1 \leftrightarrow y_1) \wedge (x_2 \leftrightarrow y_2)$ with the order in the variables $x_1 < y_1 < x_2 < y_2$



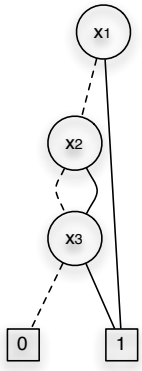
Examples of BDD

BDD for 1

BDD for 1 with one redundant test (and one removed from the preceding example)



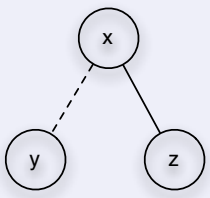
BDD for 1 with two redundant tests



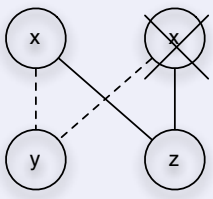
BDD for $x_1 \vee x_3$ with one redundant test

ROBDD (Reduced Order Binary Decision Diagram) (or just BDD)

Constraints to have a correct ROBDD



$x < y$ and $x < z$ (1 and 0 are greater than any variables)



Nodes must be unique

Only non-redundant test must be present

Binary Decision Diagram

Binary Decision Diagram (BDD)

Given ordered variables $X = (x_1, x_2, \ldots, x_n)$, a BDD is a rooted, directed, acyclic graph (V, E) with

- one or two terminal nodes of out-degre zero labeled 0 or 1 (0 and 1 if both are present)
- $v \in V \setminus \{0, 1\}$ are non-terminal vertices with out-degre two and has attributes
  - $var(v) \in X$
  - $low(v) \in V$
  - $high(v) \in V$
- $\forall u, v \in V$ with $v = low(u)$ or $v = high(u)$: $var(u) < var(v)$
- $var(u) = var(v), low(u) = low(v), high(u) = high(v)$ implies $u = v$
- $low(u) \neq high(u)$

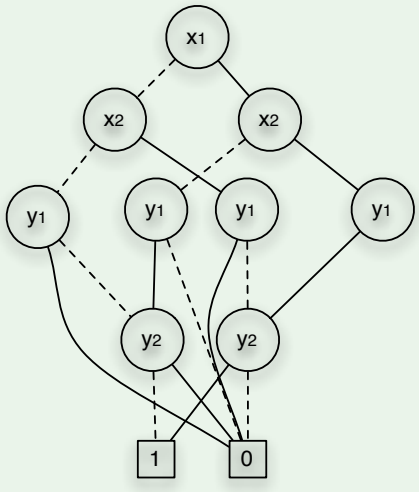
Properties of ROBDD

Canonicity

For a given order in $x_1, x_2, \ldots x_n$ there is a unique ROBDD for a given formula (or any equivalent formula)

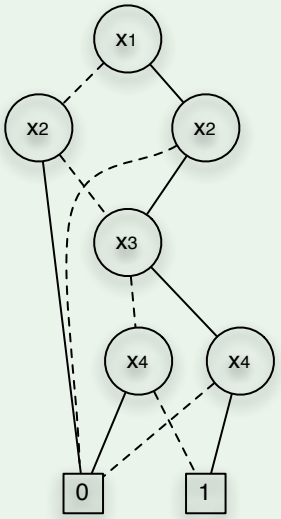
Depending on the order in the variables the ROBDD of a formula can have a very different size

ROBDD for $x_1 \leftrightarrow y_1 \wedge x_2 \leftrightarrow y_2$ with the order $x_1 < x_2 < y_1 < y_2$



Constructing and manipulating ROBDDs

ROBDD for $x_1 \leftrightarrow x_2 \wedge x_3 \leftrightarrow x_4$ with the order $x_1 < x_2 < x_3 < x_4$



Implementation with an array

u	*var*	*low*	*high*
0	5		
1	5		
2	4	1	0
3	4	0	1
4	3	2	3
5	2	4	0
6	2	0	4
7	1	5	6

Building a ROBDD

Makenode(H, max, b, i, ℓ, h) : adding a node if it does not exist yet

Requirement for efficiency

- a hash table $H : (i, \ell, h) \mapsto u$;
- *member*(H, i, ℓ, h) true iff $(i, \ell, h) \in H$:
- *lookup*(H, i, ℓ, h) return the position u of (i, ℓ, h) in b ;
- *insert*(H, i, ℓ, h, u) insert in H for (i, ℓ, h) its position u in b

Building a ROBDD

Makenode(H, max, b, i, ℓ, h) : returns the "good" node (added if needed)

Require: $H : (i, \ell, h) \mapsto u$,
Require: b the BDD in construction,
Require: max its current size,
Ensure: adds (i, ℓ, h) in b if needed and returns its position in b
  if $\ell = h$ **then**
    return ℓ
  else if *member*(H, i, ℓ, h) **then**
    return *lookup*(H, i, ℓ, h)
  else
    $max \leftarrow max + 1$
    $b.var(max) \leftarrow i$
    $b.low(max) \leftarrow \ell$
    $b.high(max) \leftarrow h$
    insert(H, i, ℓ, h, max)
    return max
  end if

Building a ROBDD

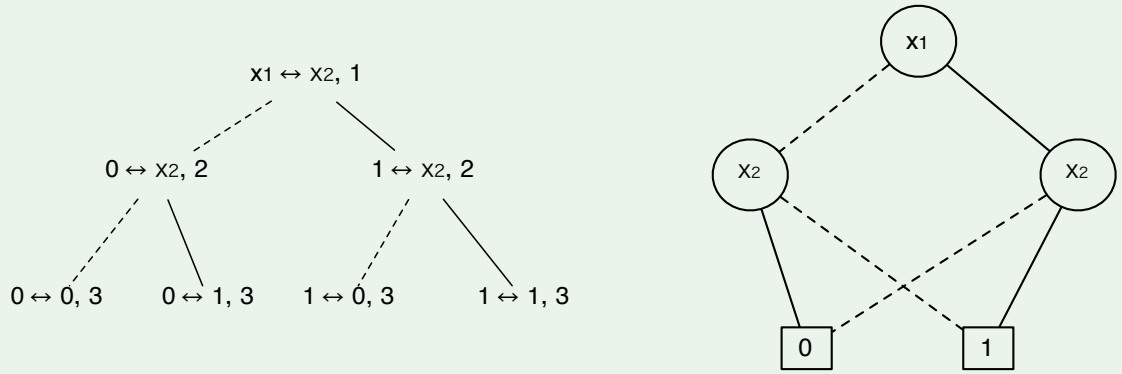
Build(t) maps a boolean expression t into a ROBDD

Ensure: build in b the ROBDD for t {Depth first and construction in postorder}
  function $build'(t, i) =$
  if $i > n$ **then**
    if $t = \bot$ **then**
      return 0
    else
      return 1
    end if
  else
    $\ell \leftarrow build'(t[0/x_i], i + 1)$ {Builds low son}
    $h \leftarrow build'(t[1/x_i], i + 1)$ {Builds high son}
    return *makenode*(H, max, b, i, ℓ, h) {Builds node (if needed)}
  end if
  end $\{build'\}$

  $H \leftarrow emptytable$
  $max \leftarrow 1$
  $b.root \leftarrow build'(t, 1)$
  return b

Build Example

Build($x_1 \leftrightarrow x_2$)



Operations on ROBDD $R = A\ op\ B$

All binary operators are implemented by the same general algorithm APPLY (op, u_1, u_2) where

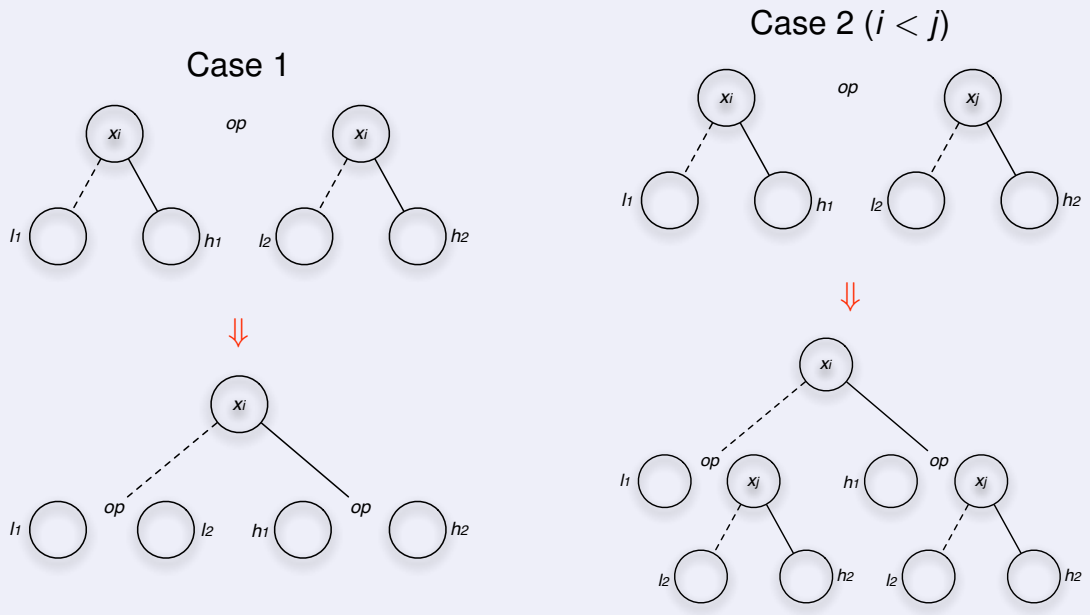
- op specifies the operator
- u_1 and u_2 are the ROBDD for the boolean expressions t^{u_1} and t^{u_2}

APPLY : 3 cases

1. $(x_i \rightarrow h_1, \ell_1)\ op\ (x_i \rightarrow h_2, \ell_2) = (x_i \rightarrow (h_1\ op\ h_2), (\ell_1\ op\ \ell_2))$
2. $x_i < x_j : (x_i \rightarrow h_1, \ell_1)\ op\ (x_j \rightarrow h_2, \ell_2) = (x_i \rightarrow (h_1\ op\ (x_j \rightarrow h_2, \ell_2)), (\ell_1\ op\ (x_j \rightarrow h_2, \ell_2))$
3. $x_i > xj$: symmetric to case 2

Operations on ROBDD $R = A\ op\ B$

Principle of APPLY



Algorithm APPLY

APPLY (op, b_1, b_2) (begin)

function app$(u_1, u_2) =$
if $G(u_1, u_2) \neq empty$ **then**
  return $G(u_1, u_2)$
else
  if $u_1 \in \{0,1\} \wedge u_2 \in \{0,1\}$ **then**
    $res \leftarrow op(u_1, u_2)$
  else if $var(u_1) = var(u_2)$ **then**
    $res \leftarrow$
    $makenode(var(u_1), app(low(u_1), low(u_2)), app(high(u_1), high(u_2)))$
  else if $var(u_1) < var(u_2)$ **then**
    $res \leftarrow makenode(var(u_1), app(low(u_1), u_2), app(high(u_1), u_2))$
  else $\{var(u_1) > var(u_2)\}$
    $res \leftarrow makenode(var(u_2), app(u_1, low(u_2)), app(u_1, high(u_2)))$
  end if
  $G(u_1, u_2) \leftarrow res$
  return res
end if

Algorithm APPLY

APPLY (op, b_1, b_2) (end)

for all $i \leq max(b_1) \wedge j \leq max(b_2)$ **do**
  $G(i,j) \leftarrow empty$
end for
$b.root \leftarrow app(b_1.root, b_2.broot)$
return b

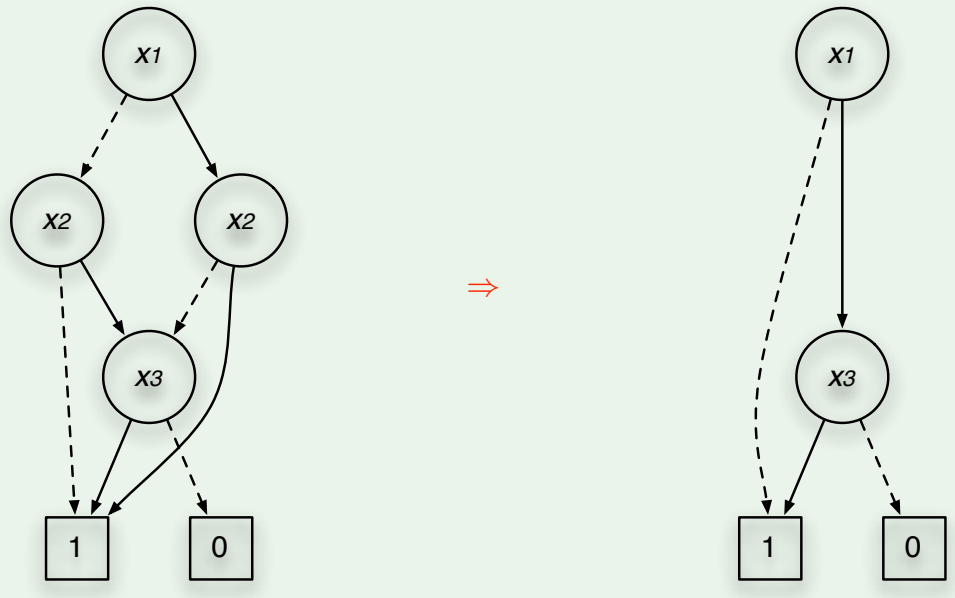
Operations on ROBDD

RESTRICT(u,j,b) (ROBDD for $u[b/x_j]$)

function $res(u)$
if $var(u) > j$ **then**
  return u
else if $var(u) < j$ **then**
  return $makenode(var(u), res(low(u)), res(high(u)))$
else if $b = 0$ **then**
  return $res(low(u))$
else
  return $res(high(u))$
end if{res}
return $res(u)$

Example of restrict

$u[0/x_2]$

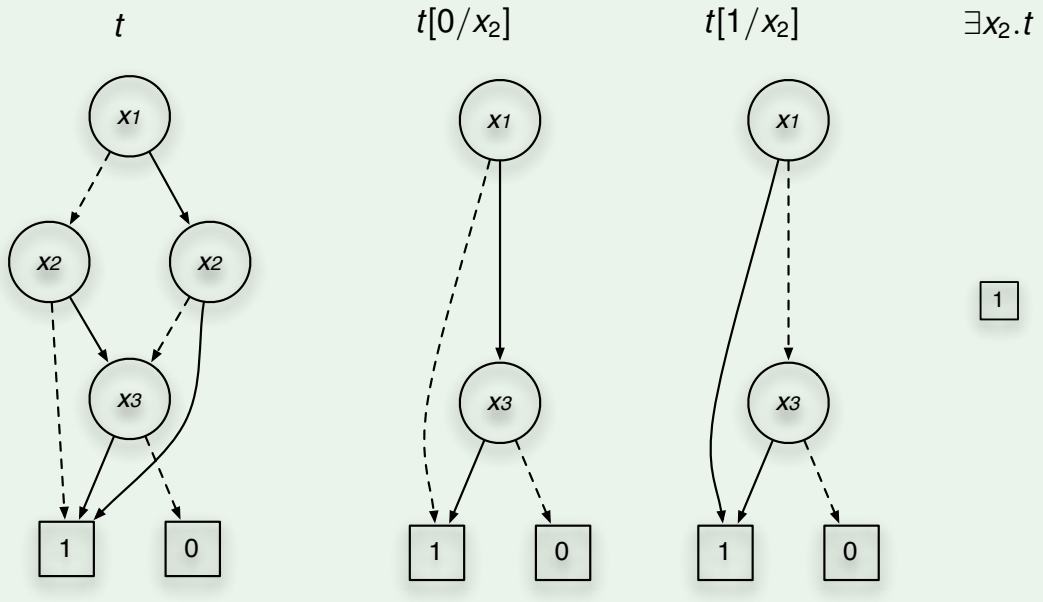


Operations on ROBDD

Existential quantification(ROBDD for $\exists x.t$)

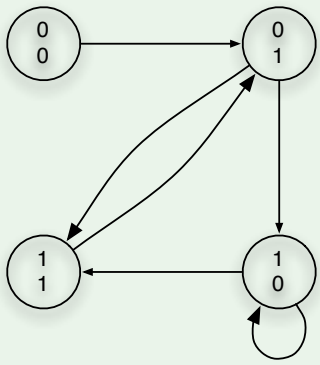
$\exists x.t = t[0/x] \vee t[1/x]$

$\exists x_2.u$



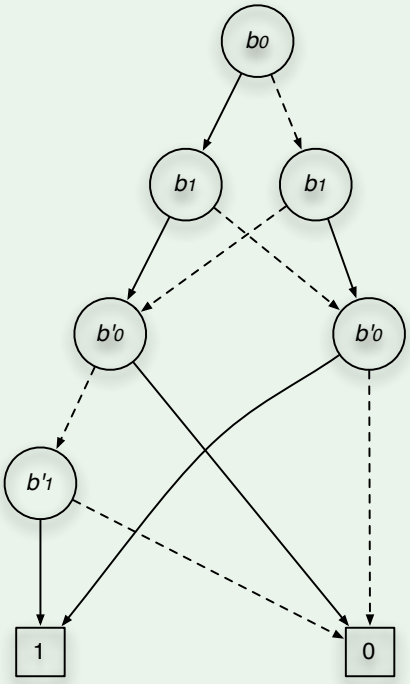
Model checking with BDD

Example of model checking with BDD



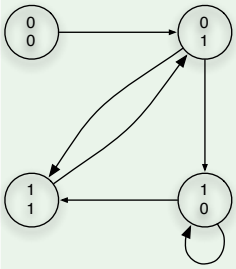
Transition relation R

b_0	0	0	0	1	1	1
b_1	0	1	1	0	0	1
b'_0	0	1	1	1	1	0
b'_1	1	1	0	1	0	1



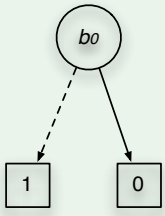
Model checking with BDD

Example : BDD of $\exists \bigcirc p = \lambda v.\exists v'.R \wedge p'$

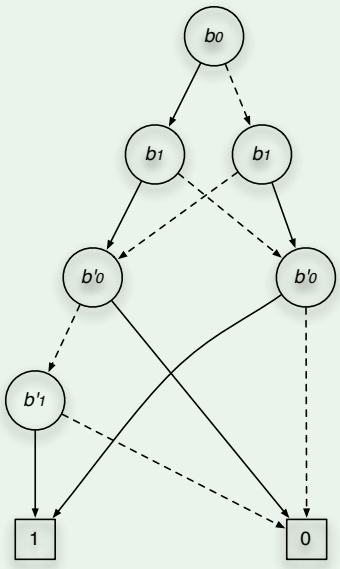


$p = \neg b_0$

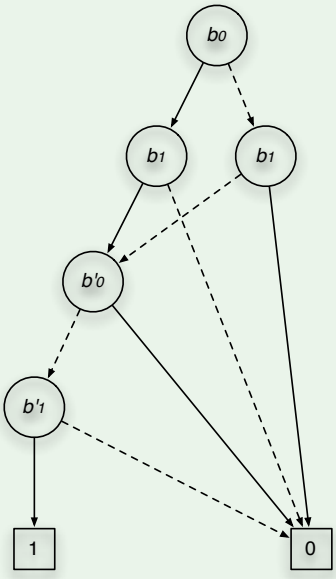
b_0	0	0
b_1	0	1



R

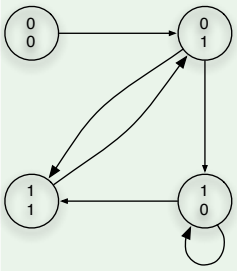


$R \wedge p'$



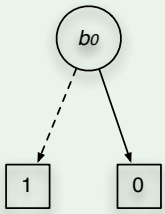
Model checking with BDD

Example : BDD of $\exists \bigcirc p = \lambda v.\exists v'.R \wedge p'$

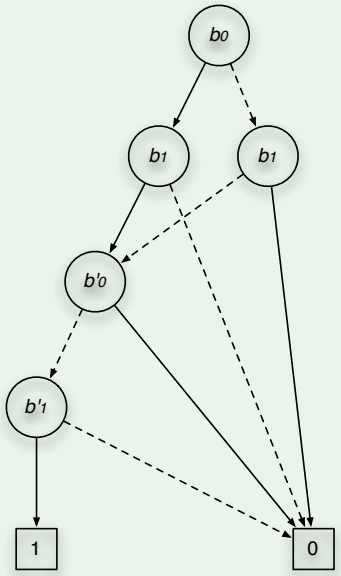


$p = \neg b_0$

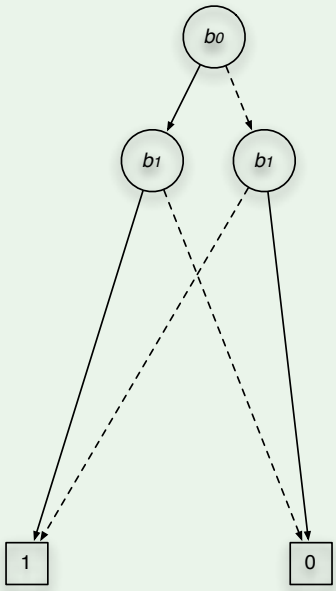
b_0	0	0
b_1	0	1



$R \wedge p'$



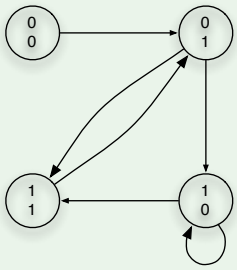
$\exists \bigcirc p = \lambda v.\exists v'.R \wedge p'$



Model checking with BDD

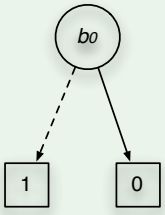
Example 2 : set of states that reach a state where p holds

$$\exists \Diamond p = \mu X.(p \vee \langle\rangle X)$$



$p = \neg b_0$

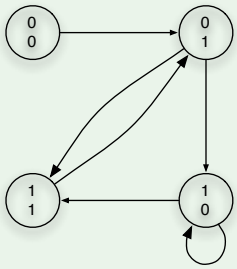
b_0	0	0
b_1	0	1



X	$p \vee \langle\rangle X$
$\varnothing$	p
p	$p \vee \langle\rangle p$
$p \vee \langle\rangle p$	$p \vee \langle\rangle(p \vee \langle\rangle p) = \top$
$\top$	$\top$

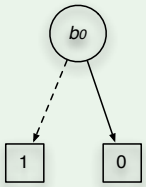
Model checking with BDD

Example 3 : BDD of $\mathrm{suc}^{\mathcal{R}}_{\exists}(p) = \lambda v'.\exists v.R \wedge p$

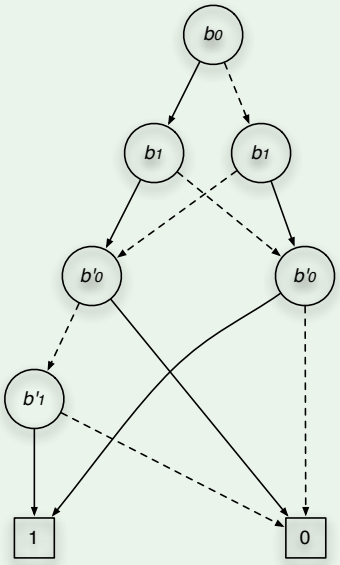


$p = \neg b_0$

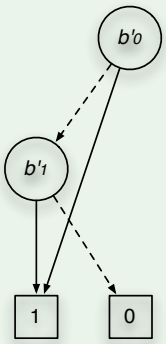
b_0	0	0
b_1	0	1



R

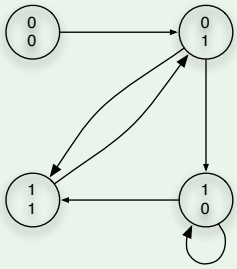


$\mathrm{suc}^{\mathcal{R}}_{\exists}(p) = \exists v.R \wedge p$



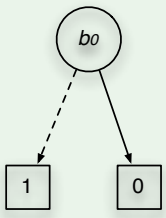
Model checking with BDD

Example : BDD of $\mathrm{suc}^{*}_{\exists}(p)$

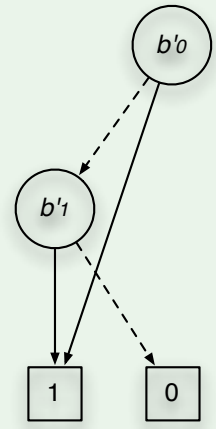


$p = \neg b_0$

b_0	0	0
b_1	0	1



$\mathrm{suc}^{\mathcal{R}}_{\exists}(p) = \exists v.R \wedge p$



$(\neg b_0 \wedge b_1) \vee b_0$

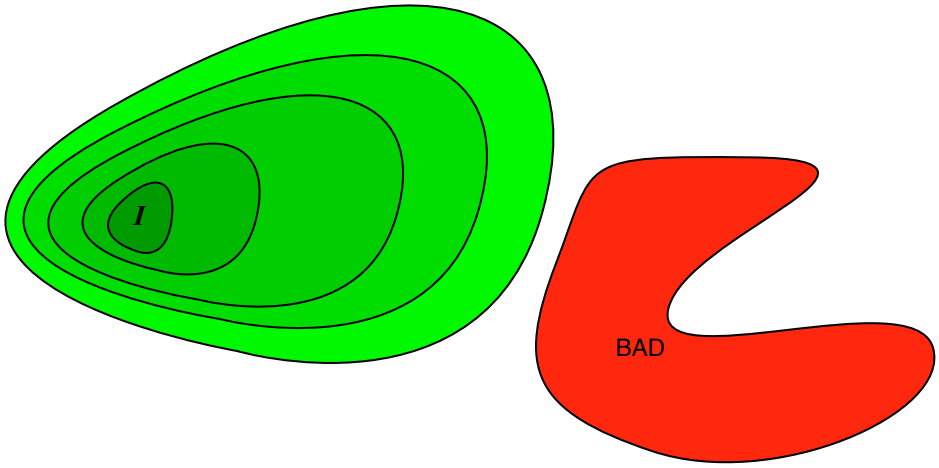
i	$\bigvee_{i=0..n} \mathrm{suc}^{i}_{\exists}(p)$
0	$\neg b_0$
1	$\top$

Checking safety properties

Nothing "BAD" can happen

$\equiv$ No bad states are reachable from an initial state

- $\mathrm{suc}^*_\exists(\mathcal{I}) \cap \mathit{BAD} = \varnothing$ (forward search)

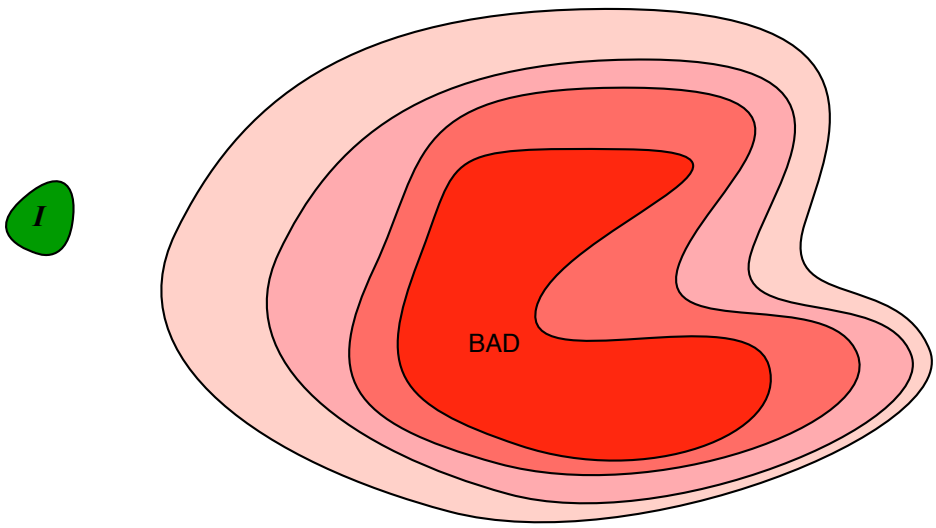


Checking safety properties

Nothing BAD can happen

$\equiv$ No bad states are reachable from an initial state

- $\mathrm{pre}^*_\exists(\mathit{BAD}) \cap \mathcal{I} = \varnothing$ (backward search)



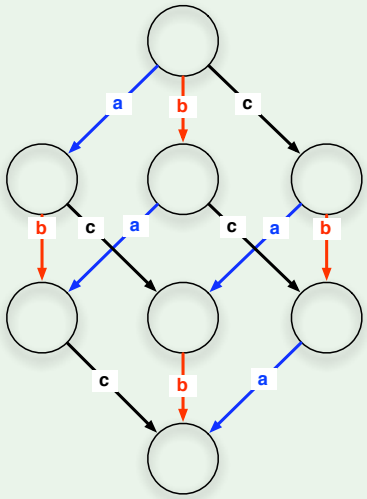
Plan

Asynchronous computation and interleaving semantics

Note

Slides done with the help of a tutorial from Edmund Clarke (see web)

Example : 3 independant events (asynchronous systems) $P = a \,||\, b \,||\, c$



With n processes : 2^n states / $n!$ orderings (exponential) = state explosion problem

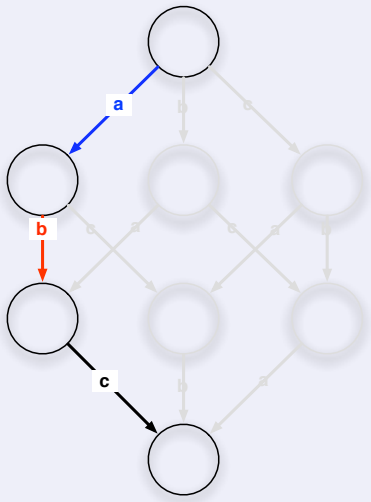
- If the temporal formula may depend on the order of the events taken, checking all interleavings is important (2^n states, $n!$ paths).
- If not, selecting any order is equivalent ($N + 1$ states, 1 path).

$\Rightarrow$ Partial order reduction : aimed at reducing the size of the state space that needs to be searched.

Partial order reduction methods

Idea

- Among the enabled events, select one to trigger first.
- A restricted graph is constructed
- The remaining subset of behaviors is sufficient to prove the property.



Formal presentation of a transition of a system

- The Kripke structure is the low level model of the system analyzed
- At a higher level, we can have concurrent systems which can have
  - Independant events (e.g. : assignments to local variables), or
  - Dependant events (e.g. assignments to shared variables, synchronizations, ...)

System = 2 concurrent processes

- $P_1 \equiv x := 1; x := 2$ endproc
- $P_2 \equiv y := 3; y := 4$ endproc
- local states of P_1 : $\{P_{10} \equiv \mathit{Initial}, P_{11} \equiv \mathit{after}(x := 1), P_{12} \equiv \mathit{after}(x := 2)$
- local states of P_2 : $\{P_{20} \equiv \mathit{Initial}, P_{21} \equiv \mathit{after}(y := 3), P_{22} \equiv \mathit{after}(y := 4)$

$\Rightarrow$ here each assignment is a "transition" : e.g. $\langle P10, P20\rangle \rightarrow \langle P11, P20\rangle$ and $\langle P10, P21\rangle \rightarrow \langle P11, P21\rangle$ is due to the transition $x := 1$

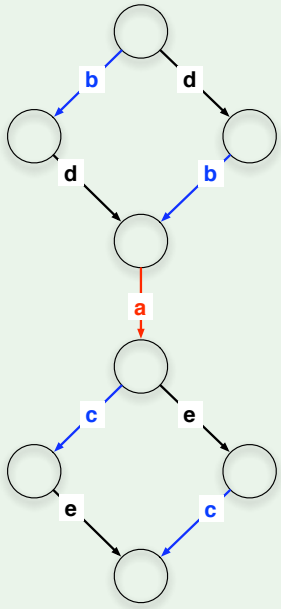
Formal presentation of a transition of a system

System = 2 concurrent processes with a “rendez-vous” (synchronization) on action a

$S \equiv P_1 |_{\{a\}} P_2$ with

- $P_1 \equiv b; a; c$ endproc
- $P_2 \equiv d; a; e$ endproc

$\Rightarrow$ a, b, c, d, e are the possible transitions of the system.



Formal presentation of a transition of a system

Therefore a “transition” must be formally seen as a binary relation between states of the Kripke structure
$\Rightarrow$ We extend the definition of Kripke structure

Definition : state transition system

$\mathcal{K} = \langle \mathcal{I}, \mathcal{S}, \mathcal{T}, \mathcal{L} \rangle$ where

- $\mathcal{T}$ is the set of transitions $\alpha \subseteq \mathcal{S} \times \mathcal{S}$
- $\mathcal{I}, \mathcal{S}, \mathcal{L}$ are defined like in “normal” Kripke structures

One transition of a state transition system can be seen as a set of transitions of the “corresponding” Kripke structure (see examples before).

Basic definitions

Basic definitions

- A transition α is enabled in a state s if there is a state s' such that $\alpha(s, s')$ holds
- Otherwise α is disabled in s.
- *enabled*(s) : set of transitions enabled in s
- A transition α is deterministic if for every state s, there is at most one s' such that $\alpha(s, s')$
- When α is deterministic we write $s' = \alpha(s)$
- A path is a finite or infinite sequence

$$\pi = s_0 \xrightarrow{\alpha_0} s_1 \xrightarrow{\alpha_1} \dots$$

  such that for every i : $\alpha_i(s_i, s_{i+1})$ holds.
- Any prefix of a path is a path
- $|\pi|$ (length of π) : number of transitions in π

Note : we only consider deterministic transitions (this is natural)

Reduced state graph

Intuition of the algorithm

- Explore (on the fly) a reduced state graph
- This can be done in depth first or breadth first search
- This can be compatible with symbolic model checking
- Since the state graph is reduced it uses less memory and takes less time

Depth first search algorithm

```
1  hash(s_0);
2  set on_stack(s_0);
3  expand_state(s_0);

4  procedure expand_state(s)
5      work_set(s) := ample(s);
6      while work_set(s) is not empty do
7          let α ∈ work_set(s);
8          work_set(s) := work_set(s) \ {α};
9          s' := α(s);
10         if new(s') then
11             hash(s');
12             set on_stack(s');
13             expand_state(s');
14         end if;
15         create_edge(s, α, s');
16     end while;
17     set completed(s);
18 end procedure
```


Depth first search algorithm

Principle of the algorithm

- Standard depth first search (DFS)
- The key point is the selection of the ample set
- If $ample(s) = enable(s)$: normal DFS
- If $ample(s) \subset enable(s)$: reduced DFS

Notes

The algorithm is “correct” if

- it terminates with a positive answer when the property holds
- it produces a counterexample otherwise

The counterexample may differ from the one obtained using the full state graph.

Ample sets

Required properties for *ample*(*s*)

1. When *ample*(*s*) is used instead of *enabled*(*s*), enough behaviors must be retained so that DFS gives correct results.
2. Using *ample*(*s*) instead of *enabled*(*s*) should result in a significantly smaller state graph.
3. The overhead in calculating *ample*(*s*) must be reasonably small.

Dependence and independence

Definitions

- An independence relation $I \subseteq \mathcal{T} \times \mathcal{T}$ is a symmetric, antireflexive relation such that for $s \in \mathcal{S}$ and $(\alpha, \beta) \in I$:
  - Enabledness : If $\alpha, \beta \in$ *enabled*(s) then $\alpha \in$ *enabled*$(\beta(s))$
  - Commutativity : $\alpha, \beta \in$ *enabled*(s) then $\alpha(\beta(s)) = \beta(\alpha(s))$

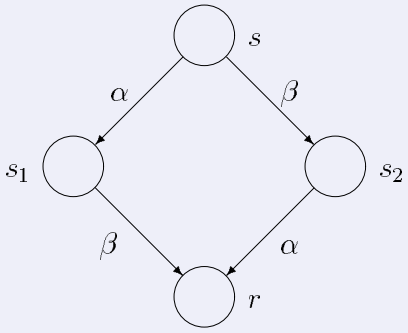
  The dependency relation D is the complement of I , namely $D = (\mathcal{T} \times \mathcal{T}) \setminus I$

Notes :

- The enabledness condition states that a pair of independent transitions do not disable one another.
- However, that it is possible for one to enable another.

Potential problems

Pseudo indenpendant transitions



If $\alpha(\beta(s)) = \beta(\alpha(s))$ two problems may occur

- Problem 1 : The checked property is sensible of the order between α and β
- Problem 2 : choosing e.g. α first can enable new transitions not possible through the path with β first

Visible and invisible transition

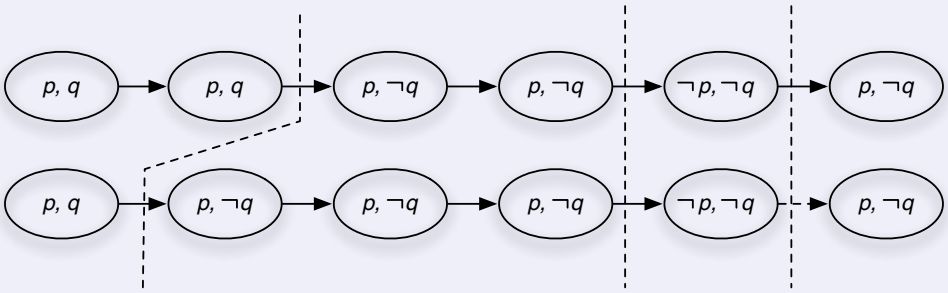
Visible and invisible transition

Given the set of propositions $\mathcal{P}$ and a subset $AP' \subseteq \mathcal{P}$

- A transition α is invisible with respect to AP' if $\forall s, s' \in S. s' = \alpha(s) \Rightarrow \mathcal{L}(s) \cap AP' = \mathcal{L}(s') \cap AP'$
- A visible transition is one not invisible

Stuttering equivalence

Stuttering equivalence



Two infinite paths $\sigma = s_0 \xrightarrow{\alpha_0} s_1 \xrightarrow{\alpha_1} \dots$ and $\rho = r_0 \xrightarrow{\beta_0} r_1 \xrightarrow{\beta_1} \dots$ are stuttering equivalent ($\sigma \sim_{st} \rho$) if there are two infinite sequences of integers

$$0 = i_0 < i_1 < i_2 < \dots \text{ and } 0 = j_0 < j_1 < j_2 < \dots$$

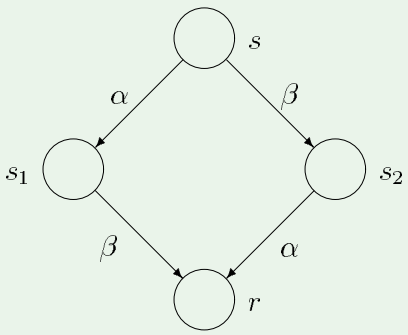
such that for every $k \geq 0$

- $L(s_{i_k}) = L(s_{i_k+1}) = \dots = L(s_{i_{k+1}-1}) = L(r_{j_k}) = L(r_{j_k+1}) = \dots = L(r_{j_{k+1}-1})$

Can also be defined for finite paths

Stuttering Equivalence

Stuttering equivalence example



- Suppose α is invisible
- $\mathcal{L}(s) = \mathcal{L}(s_1)$
- $\mathcal{L}(s_2) = \mathcal{L}(r)$

Consequently

$$s\ s_1\ r \sim_{st} s\ s_2\ r$$

(The paths $s\ s_1\ r$ and $s\ s_2\ r$ are stuttering equivalent)

LTL and stuttering equivalence

Definition : LTL formula ϕ invariant under stuttering

if and only if, for each pair of paths π and π' such that $\pi \sim_{st} \pi'$

$$\pi \models \phi \iff \pi' \models \phi$$

Definition : LTL_X

LTL without the next operator

Theorem

Any LTL_X property is invariant under stuttering

Stuttering equivalent systems

Definition : stuttering equivalent systems

Given two transition systems $\mathcal{K}_1$ and $\mathcal{K}_2$ and suppose they have initial state resp. s_0 and s'_0.
$\mathcal{K}_1$ is stuttering equivalent to $\mathcal{K}_2$ if and only if

- For each path σ of $\mathcal{K}_1$ which starts in s_0, there is a path σ' of $\mathcal{K}_2$ starting in s'_0 such that $\sigma \sim_{st} \sigma'$
- For each path σ' of $\mathcal{K}_2$ which starts in s'_0, there is a path σ of $\mathcal{K}_1$ starting in s_0 such that $\sigma \sim_{st} \sigma'$

Corrollary

For two stuttering equivalent transition systems $\mathcal{K}_1$ and $\mathcal{K}_2$ (with initial states resp. s_0 and s'_0) and every LTL_X property ϕ

$$\mathcal{K}_1 \models \phi \iff \mathcal{K}_2 \models \phi$$

DFS algorithm and ample sets

- Commutativity and invisibility will allow us to devise an algorithm which selects ample sets
- so that for every path not considered, there is a stuttering equivalent path that is considered
- Therefore, the reduced state space is stuttering equivalent to the full state space.

Definition : fully expanded state s

a state s is fully expanded when

$$ample(s) = enabled(s)$$

Construction of ample sets

Four conditions for selecting $ample(s)$ to preserve satisfaction of LTL_X formulae

Condition C0

$$ample(s) = \varnothing \iff enables(s) = \varnothing$$

Construction of ample sets

Condition **C1**

Along every path in the full state graph that starts at s :
a transition that is dependent on a transition in *ample*(s) can not be executed without one in *ample*(s) occurring first

Normally we should check **C1** on the full state space
$\Rightarrow$ we need a "way" of checking that **C1** holds without actually constructing the full state graph (see below).

Construction of ample sets

Lemma

The transitions in *enabled*(s) $\setminus$ *ample*(s) are all independent of those in *ample*(s)

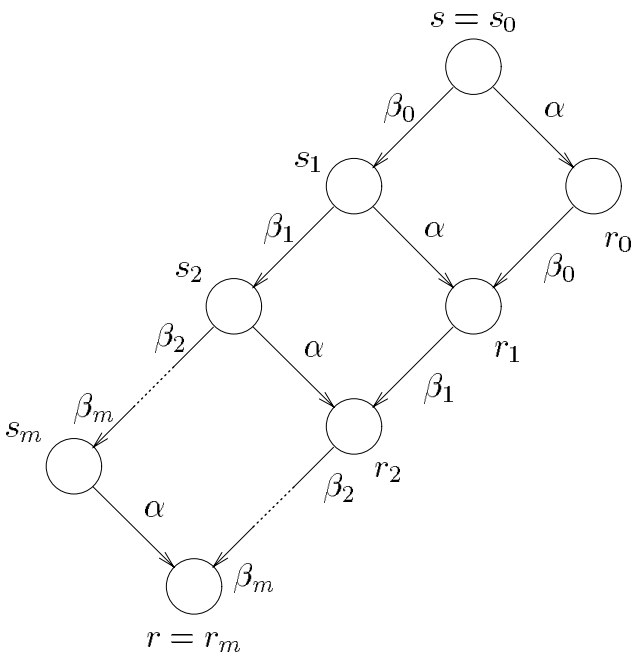
Possible forms of paths

From **C1** we can see that any path can have two possible forms

1. $\beta_0\ \beta_1\ \ldots\ \beta_m\ \alpha$ where $\alpha \in$ *ample*(s) and each β_i is independent of all transitions in *ample*(s) including α
2. An infinite sequence of $\beta_0\ \beta_1\ \ldots$ where each β_i is independent of all transition in *ample*(s)

Construction of ample sets

- Since β_i are independent from α they do not disable it
- In particular for case 1, we have



We want paths with α first stuttering equivalent to the one with α last.

Construction of ample sets

Condition C2 (invisibility)

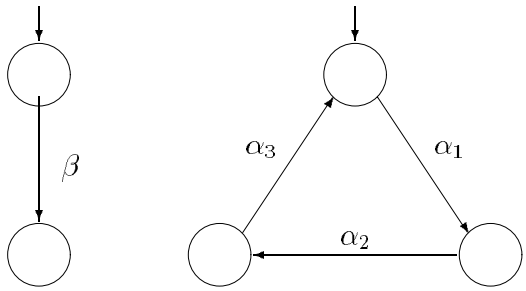
If *s* is not fully expanded then every $\alpha \in$ *ample*(*s*) is invisible

For paths of the form $\beta_0\ \beta_1\ \ldots$ that starts at *s* with no β_i in *ample*(*s*)
$\alpha\ \beta_0\ \beta_1\ \ldots$ is stuttering equivalent to $\beta_0\ \beta_1\ \ldots$

Problem with correctness condition

C1 and **C2** are not sufficient to guarantee that the reduced state graph is stuttering equivalent to the full one.
Some transition could be delayed forever

- β visible (change a proposition *p*
- β independent from invisible α_i



The process on the right performs the invisible α_i forever !

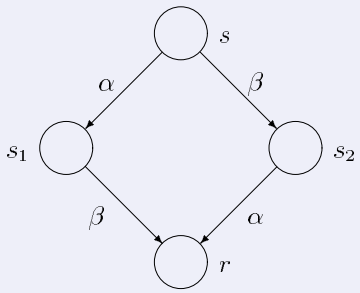
Construction of ample sets

Condition C3 (Cycle closing condition)

A cycle is not allowed if some transition β is enabled in every states in this cycle but where none of these states *s* include β in *ample*(*s*)

Have we avoided our potential problems

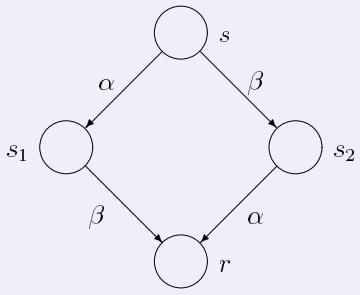
Potential problems



- Problem 1 : The checked property is sensible of the order between α and β
- Problem 2 : choosing e.g. α first can enable new transitions not possible through the path with β first

Have we avoided our potential problems ?

Potential problems



- Problem 1 : The checked property is sensible of the order between α and β

Analysis of potential problem 1

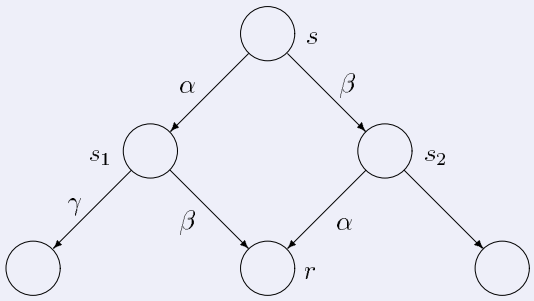
- Assume $ample(s) = \{\beta\}$
- and s_1 is not in the reduced graph
- By condition **C2**, β must be invisible

$\Rightarrow$ $s\ s_2\ r \sim_{st} s\ s_1\ r$

- We are only interested in stuttering invariant properties
- Both sequences cannot be distinguished

Have we avoided our potential problems ?

Potential problems



- Problem 2 : choosing e.g. α first can enable new transitions not possible through the path with β first

Analysis of potential problem 2

- Assume γ enabled in s_1
- γ is independent of β. Otherwise, the sequence $\alpha\ \beta$ violates **C1**
- Then, γ is enabled in r
- Assume $s_1 \xrightarrow{\gamma} s_1'$ and $r \xrightarrow{\gamma} r'$
- (β is invisible) $s\ s_1\ s_1' \sim_{st} s\ s_2\ r\ r'$

$\Rightarrow$ properties invariant under stuttering will not distinguish between the two.

Heuristic for ample sets

Model of a program

Assume that the concurrent program is composed of processes

- $pc_i(s)$: program counter of process P_i
- $pre(\alpha)$: set of transitions whose execution may enable α
- $dep(\alpha)$: set of transitions dependent of α
- T_i : set of transitions of process P_i
- $T_i(s) = T_i \cap enabled(s)$: set of transitions of process P_i enabled in s
- $current_i(s)$: set of transitions of process P_i enabled in some s' such that $pc_i(s') = pc_i(s)$

Heuristic for ample sets

Dependency relation for different models of computations

- Pairs of transitions that share a variable, which is changed by at least one of them are dependent
- Pairs of transitions belonging to the same process are dependent
- Two send transitions that use the same message queue are dependent.
- Two receive transitions that use the same message queue are dependent.

Heuristic to construct ample sets

Obvious candidate for $ample(s)$

set $T_i(s)$ (transitions enabled in s for process P_i)

- Since the transition $T_i(s)$ are interdependent, an ample set must either include all $T_i(s)$ or no transition from $T_i(s)$
- To construct $ample(s)$: start to take an non emty $T_i(s)$
- Check whether $ample(s) = T_i(s)$ satisfies condtion **C1** (see below)
- If $T_i(s)$ is not good : take another non empty $T_j(s)$ (and hope to find a good one)

Heuristic to construct ample sets

Two cases where $ample(s) = T_i(s)$ violates **C1**

The problem occurs when a transition α, interdependent to transitions in $T_i(s)$, is enabled.
Possible causes

1. α belongs to process P_j with $(j \neq i)$: a necessary condition is that $dep(T_i(s)) \cap T_j \neq \varnothing$ (can be checked effectively)
2. α (the first transition that violates **C1**) belongs to process P_i $(\alpha \in T_i)$
   - Suppose α is executed from state s'
   - The path between s and s' are independent of $T_i(s)$, and hence from other processes
   - Therefore $pc_i(s') = pc_i(s)$ and $\alpha \in current_i(s)$
   - $\alpha \notin T_i(s)$
   - $\alpha \in current_i(s) \setminus T_i(s)$
   - (α disabled in s ; enabled in s') : $\exists \beta \in pre(\alpha)$ in the sequence from s to s'
   - Thus, a necessary condition is that $\exists j \neq i.pre(current_i(s) \setminus T_i(s)) \cap T_j \neq \varnothing$ (can be checked effectively)

Plan

Example of symmetry reduction

Example in B

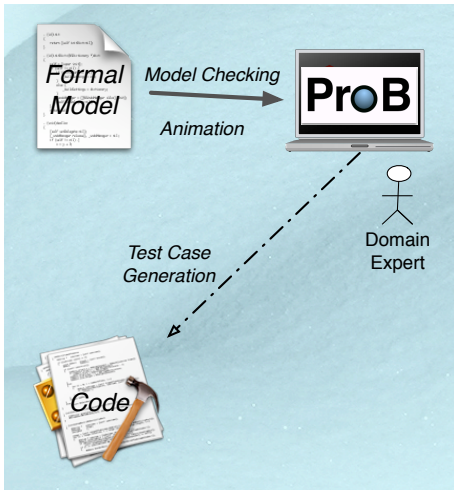
The B-method [Abr96] is

- A **language** to write high level specifications of software systems with properties (invariant) they must satisfy
- A **refinement method** to design system
- A development **environment with theorem proving tools** to prove the invariants and refinements are valid and obtain code

Critique

- It is very **difficult to write** a complete formal specification and derive the code
- It is difficult for non formalists to read and **understand** the specifications
- Specifications may be wrong (even with correct proofs) !

Some solution : PROB animator & model-checker



PROB

- allows a quick validation and debug of the models
- make the models comprehensible to domain expert
- allows people to build partial specifications

Features

- **Animation** and **model checking tool**
- kernel written in prolog
- Applied successfully to industrial examples (Volvo, Nokia, Clearsy, . . .)

PROB and the State explosion problem

State space to analyse may have an exponential size ⇒

Possible Model Checking reduction techniques

- Symbolic Model Checking
- Partial order reduction
- Symmetry reduction **(promising)**
- ...

General Goal

Symmetry reduction techniques
to model check
B specifications

Basic principle

Work with a quotient state space of the system (modulo symmetry equivalence)
Linked with the isomorphism problem Example

Motivation

Sometimes Symmetry reduction is not enough

- Hard problem (see Wikipedia : graph isomorphism for more information)
- For some practical examples too expensive

Our work

⇒ **Alternative solutions ?**

⇒ **Approximate methods ?**

Efficient approximate analysis method
with symmetry reduction

B in a nutshell

Basic concepts : set theory with predicate logic

- Logical predicates
- basic datatypes : integer, natural, ...
- Pairs ($x \mapsto y$)
- Given sets : explicitely enumerated
- **Deferred sets** : elements not given a priori

Relations, functions, ...

- $\mathrm{dom}(x)$, $\mathrm{ran}(x)$, image ($r[S]$), , inverse (R^{-1}), composition ($R0$; $R1$), restrictions ($U \lhd R$, $U \lhd\!\!\!\!- R$, $R \rhd U$, $R \rhd\!\!\!\!- U$), ...
- partial function ($x \pfun y$), total function ($x \fun y$), injection ($x \pinj y$, $x \inj y$), surjection ($x \psur y$, $x \sur y$), bijection ($x \bij y$)
- sequences, records, trees, ...

Operations

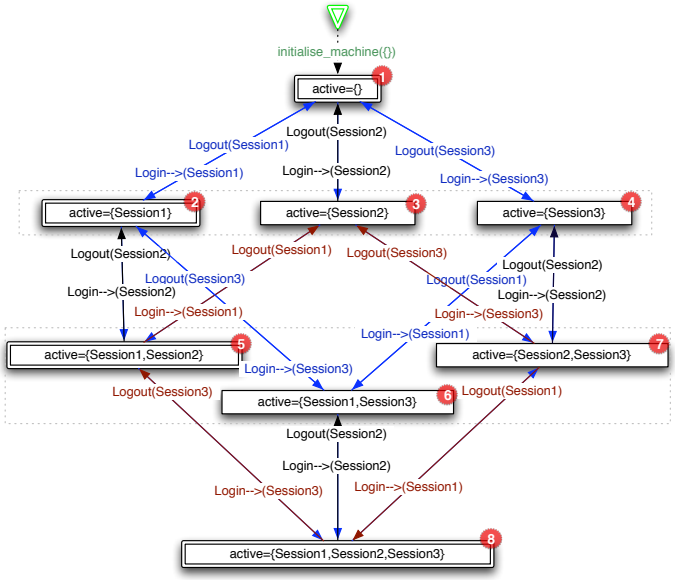
- Transform the state of a machine
- Must preserve the invariant

A simple login

Simple login

MACHINE *LoginVerySimple*
SETS *Session*
VARIABLES *active*
INVARIANT $active \subseteq Session$
INITIALISATION $active := \varnothing$
OPERATIONS
$res \leftarrow Login =$ **ANY** s **WHERE** $s \in Session \wedge s \notin active$ **THEN**
$res := s \parallel active := active \cup \{s\}$ **END**;
$Logout(s) =$ **PRE** $s \in active$ **THEN**
$active := active - \{s\}$ **END**
END

Instantiate the deferred sets : e.g. *Session* = 3



Symmetry

Informally, two states are symmetric - the invariant has the same truth value in both states, - both can execute the same sequences of operations (possibly up to some renaming of data values in the parameters)

Back to introduction

In practice to be efficient

- On the fly analysis
- Does not keep the state space

Second running example : the dining philosopher

Dining Philosopher (without protocol)

MACHINE *Philosophers*
SETS *Phil*; *Forks*
CONSTANTS *lFork*, *rFork*
PROPERTIES
$lFork \in Phil \rightarrowtail\!\!\!\!\!\rightarrow Forks \ \wedge \ rFork \in Phil \rightarrowtail\!\!\!\!\!\rightarrow Forks \wedge$
$card(Phil) = card(Forks) \ \wedge \ \forall pp.(pp \in Phil \Rightarrow lFork(pp) \neq rFork(pp)) \wedge$
$\forall st.(st \subset Phil \wedge st \neq \varnothing \ \Rightarrow rFork^{-1}[lFork[st]] \neq st)$
VARIABLES *taken*
INVARIANT
$taken \in Forks \nrightarrow Phil \wedge$
$\forall xx.(xx \in dom(taken) \Rightarrow (lFork(taken(xx)) = xx \ \vee \ rFork(taken(xx)) = xx))$
INITIALISATION $taken := \varnothing$
OPERATIONS
TakeLeftFork$(p, f) =$
PRE $p \in Phil \ \wedge \ f \in Forks \ \wedge f \notin dom(taken) \ \wedge \ lFork(p) = f$
THEN $taken(f) := p$ **END**;
TakeRightFork$(p, f) =$
PRE $p \in Phil \ \wedge \ f \in Forks \ \wedge f \notin dom(taken) \ \wedge \ rFork(p) = f$
THEN $taken(f) := p$ **END**;
DropFork$(p, f) =$
PRE $p \in Phil \ \wedge \ f \in Forks \ \wedge \ f \in dom(taken) \ \wedge \ taken(f) = p$
THEN $taken := f \lhd taken$ **END**
END

Symmetry and deferred sets

Observations

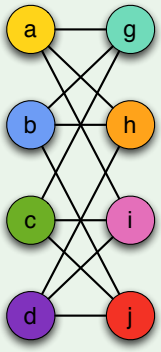
- **Elements of deferred sets** are not specified a priori and have **no name or identifier**.
- Inside a B machine one **cannot select a particular element of such deferred sets**.
- for any state of B machine, **permutations of elements inside the deferred sets preserve**
  - the truth value of B predicates and the invariant
  - the structure of the transition relation [LBST07].

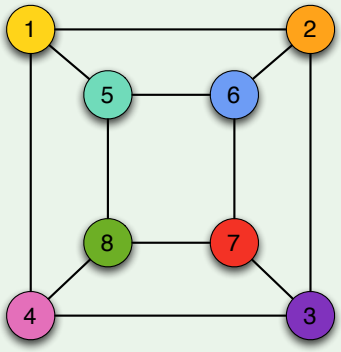
Graph Canonicalisation

Graph Canonicalisation

- **Orbit problem** : decide if two states are symmetric
- Tightly linked to detecting **graph isomorphisms** (after converting states into graphs)
- Currently has no known polynomial algorithm.
- Most efficient general purpose graph isomorphism program : ***nauty***

Example of isomorphic graphs



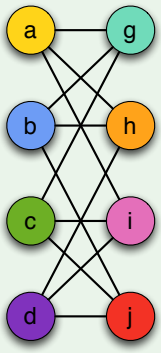


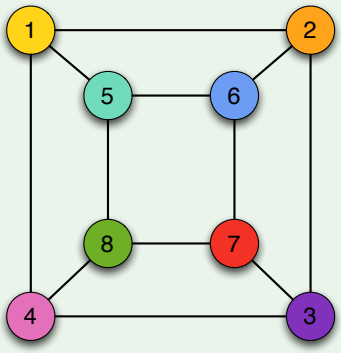
Isomorphism

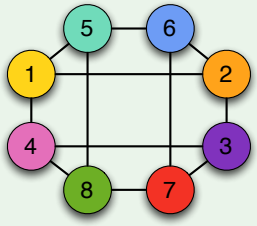
$f(a) = 1$
$f(b) = 6$
$f(c) = 8$
$f(d) = 3$
$f(g) = 5$
$f(h) = 2$
$f(i) = 4$
$f(j) = 7$

Example of isomorphic graphs

Example of isomorphic graphs

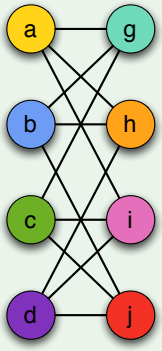






Canonical form of a graph

coding the graph



Isomorphism

$c(a) = 0$
$c(b) = 1$
$c(c) = 2$
$c(d) = 3$
$c(g) = 4$
$c(h) = 5$
$c(i) = 6$
$c(j) = 7$

Adjacency matrix

0	0	0	0	1	1	1	0
0	0	0	0	1	1	0	1
0	0	0	0	1	0	1	1
0	0	0	0	0	1	1	1
1	1	1	0	0	0	0	0
1	1	0	1	0	0	0	0
1	0	1	1	0	0	0	0
0	1	1	1	0	0	0	0

$G = 00001110000011010000101100000111111000001101000010110000011100000$

- Example of canonical form : the order which gives the smallest encoding ;
- $n!$ possible orderingS.

When symmetry is not efficient enough : symmetry markers

Analysis without symmetry : Holzmann's bitstate hashing [Hol88]

- Approximate verification technique
- Computes a **hash value for every reached state**
- State with the same hash value is not analysed any further
- **Ideal hashing function** : two different values for two different states
- In practice
  - **collisions** : some reachable states are not checked (not exhaustive analysis)
  - **very efficient**

Same idea with symmetry ?

- Hashing function **invariant to symmetry**
- replace hashing function by **marker**
- Two symmetric states have the same marker
- **Efficient** computing of the marker
- **Possible collisions : minimise their number**

Definition of Marker of a state s

Main idea

- State s seen as a graph
- Marker $m(s)$ expresses the structure of s
- Two symmetric states $\longrightarrow$ same marker
- The other way may be wrong (collision)

Everything completely identified except elements of deferred sets.

$\Rightarrow$ **Must compute markers of elements d of deferred sets**.

Markers for elements of deferred sets

Existing vertex invariant of the corresponding graph

- Number of incoming edges
- Number of outgoing edges
- ...

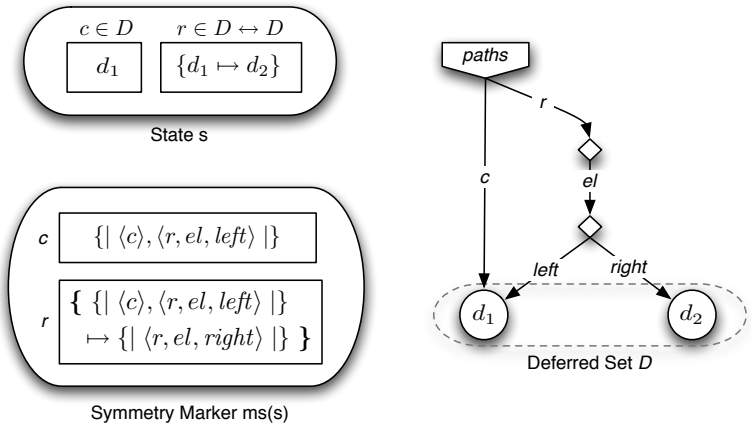
$\Rightarrow$ **find something more precise and still efficient.**

Marker of an element d of a deferred set

- must include the set of places where it is used
- $\Rightarrow$ **Compute multiset of paths leading to d in the current state**

Efficiency :
Worst case $\mathcal{O}(n^2)$ (n = nb of vertices in the graph of the global state)

Definition of Markers



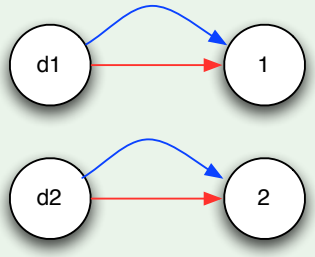
Proposition 1

Let s_1, s_2 be two states. If s_1 and s_2 are permutation states of each other then $m(s_1) = m(s_2)$.

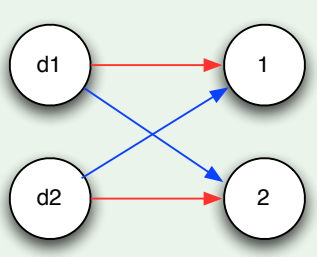
Example of what our markers can distinguish

with the deferred set $\mathcal{D} = \{d_1, d_2\}$ and variables x, y

$s_1 = \langle \{d_1 \mapsto 0\}, \{d_1\} \rangle$, $s_3 = \langle \{d_2 \mapsto 0\}, \{d_2\} \rangle$	$m(s_1) = m(s_3)$	√
$s_1 = \langle \{d_1 \mapsto 0)\}, \{d_1\} \rangle$, $s_2 = \langle \{d_2 \mapsto 0)\}, \{d_1\} \rangle$	$m(s_1) \neq m(s_2)$	√
$s_4 = \langle \{d_1 \mapsto 1, d_2 \mapsto 2\}, \{d_1 \mapsto 1, d_2 \mapsto 2\} \rangle$, $s_5 = \langle \{d_1 \mapsto 2, d_2 \mapsto 1\}, \{d_1 \mapsto 1, d_2 \mapsto 2\} \rangle$	$m(s_4) \neq m(s_5)$	√



s4

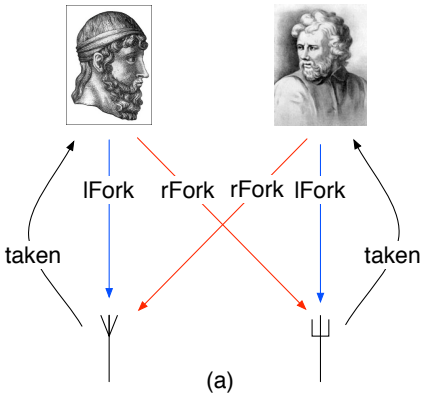


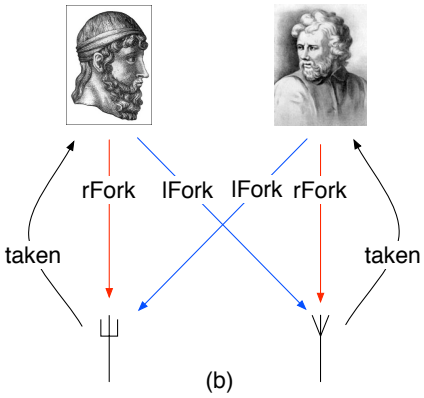
s5

Example of what our markers can and cannot distinguish

Two dining philosophers

$s_1 = p_1$ leftTakes(f) , $s_2 = p_1$ rightTakes(f)	$m(s_1) \neq (s_2)$	✓
Each has one fork (see figures)	$m(s_1) = m(s_2)$	✗





When are symmetry markers precise

Proposition 2

If each value v in s_1 and s_2 is either :

- a value not containing any element from one of the sets $D_1, \ldots, D_i$, or
- a value not containing a set, or
- a set of values $\{x_1, \ldots, x_n\} \subseteq D_k$ for some $1 \leq k \leq i$, or
- a set of pairs $\{x_1 \mapsto y_1, \ldots, x_n \mapsto y_n\}$ such that either all x_i are in *NonSym* and all y_i are elements of some deferred set D_j, or all x_i are in *NonSym* and all y_i are elements of some deferred set D_j.

Then $m(s_1) = m(s_2)$ implies that there exists a permutation function f over $\{D_1, \ldots, D_i\}$ such that $f(s_1) = s_2$.

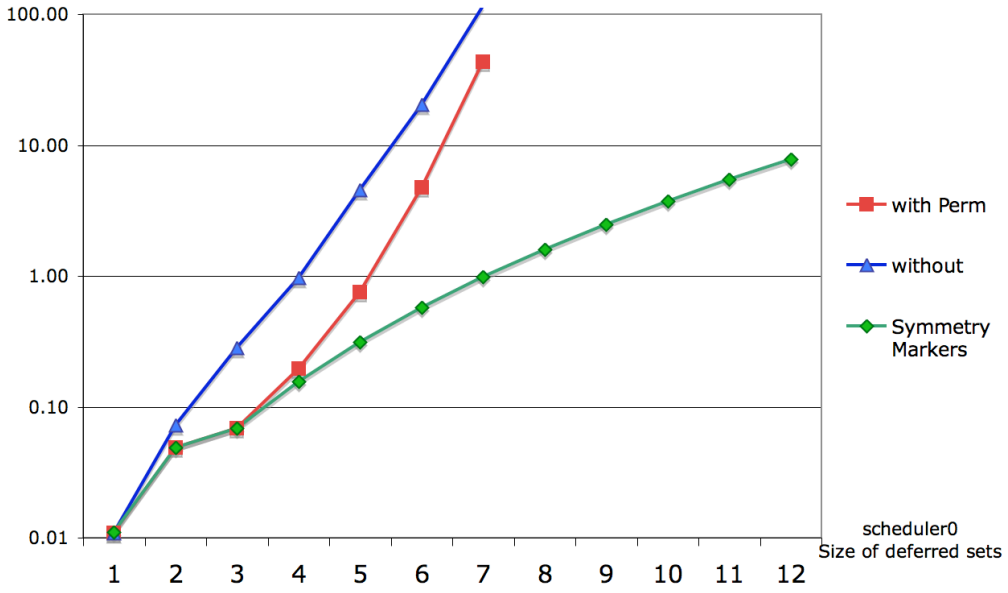
$\Rightarrow$ In that case our symmetry marker method provides a full verification.

In practice covers a lot of cases.

Empirical Evaluation

Machine	Card	Model Checking Time			Number of Nodes			Speedup over	
		wo	flood	markers	wo	flood	markers	wo	flood
Russian	1	0.05	0.05	0.05	15	15	15	1.04	1.04
	2	0.32	0.21	0.21	81	48	48	1.51	0.97
	3	1.32	0.46	0.34	441	119	119	3.92	1.35
	4	8.73	1.90	0.89	2325	248	248	9.81	2.13
	5	**54.06**	**12.18**	**2.05**	11985	459	459	26.35	5.94
scheduler0	1	0.01	0.01	0.01	5	5	5	0.98	0.99
	2	0.07	0.05	0.05	16	10	10	1.59	1.06
	3	0.28	0.07	0.06	55	17	17	4.60	1.12
	4	0.98	0.20	0.14	190	26	26	7.15	1.43
	5	4.52	0.75	0.27	649	37	37	16.87	2.81
	6	20.35	4.74	0.48	2188	50	50	42.60	9.93
	7	**114.71**	**43.47**	**0.80**	7291	65	65	143.61	54.43
scheduler1	1	0.01	0.01	0.01	5	5	5	1.09	1.12
	2	0.05	0.06	0.05	27	14	14	1.12	1.26
	3	0.41	0.11	0.09	145	29	29	4.50	1.17
	4	2.96	0.34	0.18	825	51	51	16.62	1.93
	5	23.93	1.70	0.37	5201	81	81	64.24	4.56
	6	192.97	13.37	0.70	37009	120	120	275.75	19.10
	7	**941.46**	**167.95**	**1.22**	297473	169	169	771.39	137.61
Peterson	2	0.28	0.28	0.15	49	27	27	1.87	1.89
	3	8.80	2.00	1.73	884	174	174	5.08	1.16
	4	861.49	60.13	20.66	22283	1134	1134	41.69	2.91
Philosophers	2	0.11	0.05	0.04	21	**8**	**7**	3.02	1.30
	3	1.56	0.15	0.05	337	**13**	**11**	28.83	2.80
	4	123.64	5.99	0.15	11809	**26**	**20**	799.36	38.73
Towns	1	0.01	0.01	0.01	3	3	3	1.03	1.00
	2	0.37	0.33	0.34	17	11	11	1.08	0.97
	3	63.95	12.78	12.95	513	105	105	4.94	0.99
USB	1	0.21	0.20	0.22	29	29	29	0.96	0.90
	2	8.42	4.74	6.17	694	355	355	1.36	0.77
	3	605.25	277.59	232.93	16906	3013	3013	2.60	1.19

Comparison of execution time



Model Checking time (in seconds) for scheduler0 ; log scale

Related works

Symmetry detection (Generally, specified by hand)

- Ip and Dill [ID96] : scalarset : tool Murϕ [DDHY92].
- Clarke, Jha et al [CEFJ96, Jha96] data symmetry with BDD
- extension of scalarset : untimed [BDH02, DMC05] and timed [HBL$^+$03]
- Emerson & Sistla [ES96, ES95] and tool SMC [SGE00]

$\Rightarrow$ **In B : symmetry arises naturally with the deferred sets**

Efficient identification of equivalent states

- Vertex invariants in the tool Nauty
- already discussed in [ES96] : very simple hashing function invariant to symmetry

$\Rightarrow$ **To our knowledge, the first elaborate approach and evaluation of an efficient approximation method.**

Plan

Bounded model checking

Note

These slides are the chapter 1 of a tutorial given in ACSD'06 - ATPN'06 by Keijo Heljanko and Tommi Junttila.

See : http ://www.tcs.hut.fi/ kepa/bmc-tutorial.html

Advanced Tutorial on Bounded Model Checking (BMC)

ACSD'06 - ATPN'06

26th of June 2006

Keijo Heljanko and Tommi Junttila
Keijo.Heljanko@tkk.fi, Tommi.Junttila@tkk.fi

Organisers

- D.Sc. (Tech.), Academy Research Fellow
  Keijo Heljanko
  - Email: Keijo.Heljanko@tkk.fi
  - Homepage: http://www.tcs.tkk.fi/~kepa/
- D.Sc. (Tech.) Tommi Junttila
  - Email: Tommi.Junttila@tkk.fi
  - Homepage: http://www.tcs.tkk.fi/~tjunttil/
- Our affiliation:
  Laboratory for Theoretical Computer Science,
  Helsinki University of Technology (TKK)

Thanks

Thanks to co-authors on papers related to bounded model checking (in alphabetic order):

- Armin Biere, Johannes Kepler University of Linz
- Toni Jussila, Johannes Kepler University of Linz
- Timo Latvala, University of Illinois at Urbana-Champaign
- Ilkka Niemelä, Helsinki University of Technology (TKK)
- Jussi Rintanen, National ICT Australia Limited (NICTA)
- Viktor Schuppan, ETH Zürich

Tutorial Homepage

- All the material of the Tutorial is available as PDF files from the tutorial homepage:
  http://www.tcs.tkk.fi/~kepa/bmc-tutorial.html
- The PDF files also contain lots of hyperlinks to referenced papers, tools, etc.

Software failures

Software is used widely in many applications where a bug in the system can cause large damage:

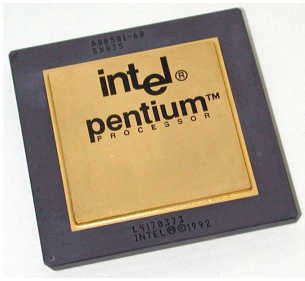
- Safety critical systems: airplane control systems, medical care, train signalling systems, air traffic control, etc.
- Economically critical systems: e-commerce systems, Internet, microprocessors, etc.

Price of Software Defects

Two very expensive software bugs:

- Intel Pentium FDIV bug (1994, approximately $500 million).
- Ariane 5 floating point overflow (1996, approximately $500 million).

Pentium FDIV - Software bug in HW



$$4195835 - ((4195835 / 3145727) * 3145727) = 256$$

The floating point division algorithm uses an array of constants with 1066 elements. However, only 1061 elements of the array were correctly initialised.

Ariane 5

Exploded 37 seconds after takeoff - the reason was an overflow in a conversion of a 64 bit floating point number into a 16 bit integer.

Finding Bugs in Concurrent Systems

The principal methods for the validation of complex parallel and distributed systems are:

- Testing (using the system itself)
- Simulation (using a model of the system)
- Deductive verification (mathematical (manual) proof of correctness, in practice done with computer aided proof assistants/theorem provers)
- Model Checking ($\approx$ exhaustive testing of a model of the system)

Why is Testing Hard?

Testing should always be done! However, testing parallel and distributed systems is not always cost effective:

- Testing concurrency related problems is often done only when rest of the system is in place
  $\Rightarrow$ fixing bugs late can be very costly.
- It is labour intensive to write good tests.
- It is hard if not impossible to reproduce bugs due to concurrency encountered in testing.
  - Did the bug-fix work?
- Testing can only prove the existence of bugs, not their non-existence.

Simulation

The main method for the validation of hardware designs:

- When designing new microprocessors, no physical silicon implementation exists until very late in the project.
- Example: Intel Pentium 4 simulation capacity (Roope Kaivola, talk at CAV05):
  - 8000 CPUs
  - Full chip simulation speed 8 Hz (final silicon > 2 GHz).
  - Amount of real time simulated before tape-out: around 2 minutes.

Deductive Verification

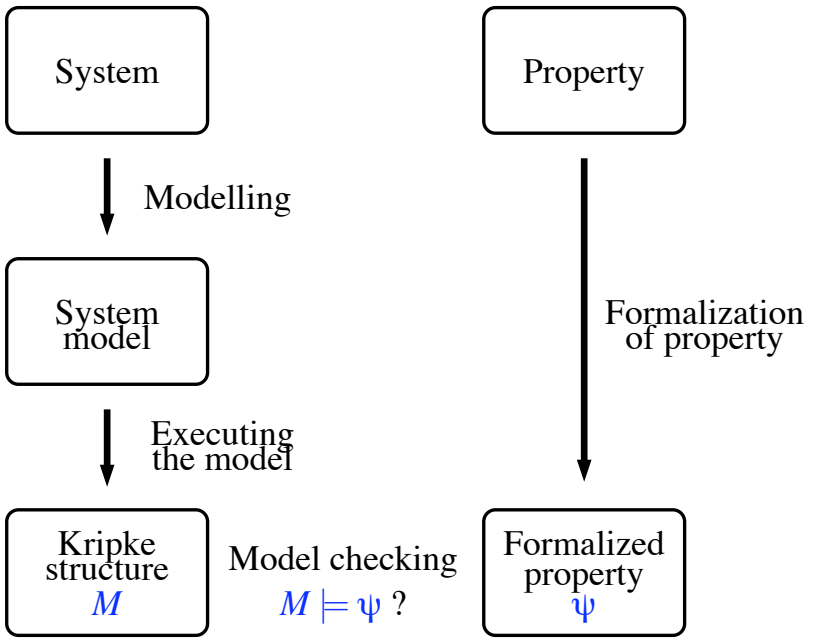
- Proving things correct by mathematical means (mostly invariants + induction).
- Computer aided proof assistants/theorem provers used to keep you honest and to prove sub-cases.
- Very high cost, requires highly skilled personnel:
  - Only for truly critical systems.
  - HW examples: Pentium 4 FPU, Pentium 4 register rename logic (Roope Kaivola: 2 man years, 2 'time bomb' silicon bugs found - thankfully masked by surrounding logic)

Model Checking

In model checking every execution of the model of the system is simulated obtaining a Kripke structure M describing all its behaviours. M is then checked against a property ψ:

- Yes: The system functions according to the specified property (denoted $M \models \psi$).
  The symbol $\models$ is pronounced "models",
  hence the term model checking.
- No: The system is incorrect (denoted $M \not\models \psi$), a counterexample is returned: an execution of the system which does not satisfy the property.

Models and Properties



Benefits of Model Checking

- In principle automated: Given a system model and a property, the model checking algorithm is fully automatic.
- Counterexamples are valuable for debugging.
- Already the process of modelling catches a large percentage of the bugs: good for rapid prototyping of concurrency related features.

Drawbacks of Model Checking

- State explosion problem: Capacity limits of model checkers can be exceeded.
- Manual modelling often needed:
  - Model checker used might not support all features of the final implementation language.
  - Abstraction used to overcome capacity problems.

Model Checking in the Industry

- Microprocessor design: Several major microprocessor manufacturers use model checking methods as a part of their design process.
- Design of Data-communications Protocol Software: Model checkers have been used as rapid prototyping systems for new data-communications protocols under standardisation.
- Mission Critical Software: NASA space program is model checking code used by the space program.
- Operating Systems: Microsoft is using model checking to verify the correct use of locking primitives in Windows device drivers.

Modelling Languages

As a language describing system models we can for example use:

- Petri nets,
- labelled transition systems (LTSs) and process algebras,
- Java programs,
- UML (unified modelling language) state machines,
- Promela language (input language of the Spin model checker), and
- VHDL,Verilog, or SMV languages (mostly for HW design).

Some Model Checking Approaches

- Explicit State Model Checking: Tools include Spin, Murφ Java Pathfinder Maria, PROD, CPN Tools, CADP, etc.
- BDD based Symbolic Model Checking: Tools include NuSMV 2, VIS, Cadence SMV, etc.
- Bounded Model Checking: Tools include BMC, CMBC, NuSMV 2, VIS, Cadence SMV, etc.

Bounded Model Checking

- Originally presented in the paper: Armin Biere, Alessandro Cimatti, Edmund M. Clarke, Yunshan Zhu: Symbolic Model Checking without BDDs. TACAS 1999: 193-207, LNCS 1579.
- A closely related approach had already been used earlier to solve artificial intelligence planning problems in: Henry A. Kautz, Bart Selman: Planning as Satisfiability.Proceedings of the 10th European conference on Artificial intelligence (ECAI'92): 359-363, 1992, Kluwer.

Basics of Bounded Model Checking

- The basic idea is the following: Encode all the executions of the system M of length k into a propositional formula $||[M]||^k$.
- Conjunct this formula with a formula $||[\neg\psi]||^k$ which is satisfiable for all executions the system of length k which violate the property ψ.
- If the formula $||[M]||^k \wedge ||[\neg\psi]||^k$ is satisfiable, a counterexample has been found.
- If the formula $||[M]||^k \wedge ||[\neg\psi]||^k$ is unsatisfiable, no counterexample of length k exists.

SAT

- The propositional satisfiability problem (SAT) is one of the main instances of NP-complete problems.
- Thus no polynomial algorithms for SAT are known.
- However, there are highly efficient SAT solvers available such as zChaff and MiniSAT which are able solve many bounded model checking problems efficiently.

SAT References

- zChaff: Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, Sharad Malik: Chaff: Engineering an Efficient SAT Solver. DAC 2001: 530-535, ACM.
- MiniSAT: Niklas Eén, Niklas Sörensson: An Extensible SAT-solver. SAT 2003: 502-518, LNCS 2919.
- SATLive! - Links to SAT related events, tools, position announcements, etc.
- SAT race 2006 - In 2006 a "light weight" variant the SAT solver competition on industrial benchmarks is arranged.

Basic Setup

- For simplicity first consider the following setup:
  - As system models we consider systems whose state vector s consist of n Boolean state variables $\langle s[0], s[1], \ldots, s[n-1] \rangle$.
  - We take $k+1$ copies of the system state vector denoted by $s_0, s_1, \ldots, s_k$.
  - Let $I(s)$ be the initial state predicate of the system, and $T(s, s')$ be the transition relation both expressed as propositional formulas.

A Simplifying Assumption

- For simplicity we assume $T(s, s')$ to be be total for now, i.e., every reachable state s should have a successor s' such that $T(s, s')$ holds.
- This assumption can and will be dropped later in this tutorial.

Unrolling the Transition Relation

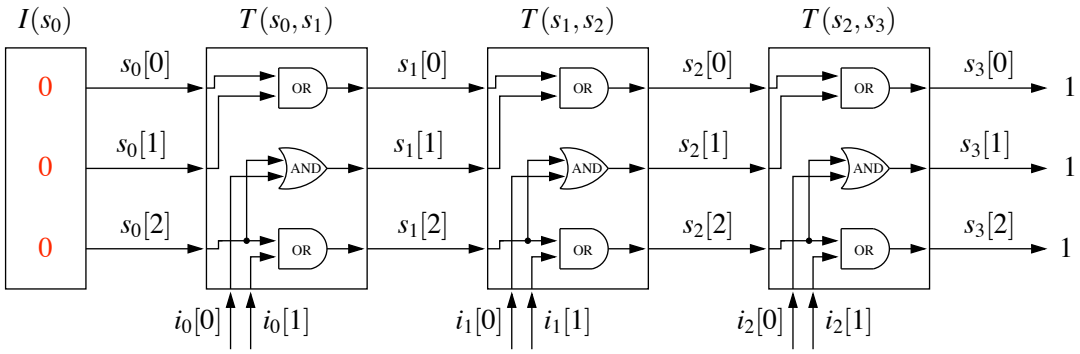
- Now the executions of the system of length k are captured by the formula:

$$|[M]|^k = I(s_0) \wedge \bigwedge_{i=1}^{k} T(s_{i-1}, s_i)$$

- For $k = 3$ this becomes:

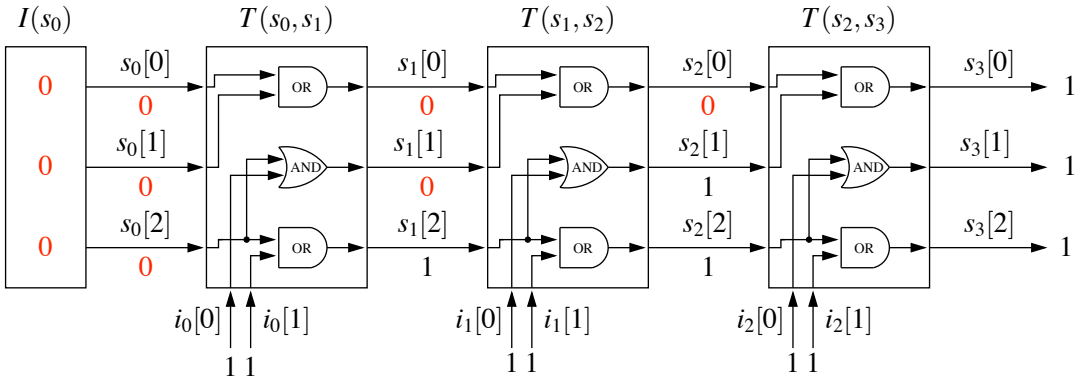
$$|[M]|^3 = I(s_0) \wedge T(s_0, s_1) \wedge T(s_1, s_2) \wedge T(s_2, s_3)$$

Circuit BMC Unrolling



What do the input vectors i_0, i_1, and i_2 need to be to reach the state $s_3 = \langle 1, 1, 1 \rangle$?

Circuit BMC Unrolling Solution



The input vectors $i_0 = \langle 1,1 \rangle$, $i_1 = \langle 1,1 \rangle$, and $i_2 = \langle 1,1 \rangle$ will reach the final state $s_3 = \langle 1,1,1 \rangle$.

Expressing Invariants

- Suppose the property ψ we want to model check is that an invariant property $P(s)$ holds for every reachable state of the system M.
- Now we get that:

$$||[\neg\psi]||^k = \bigvee_{i=0}^{k} \neg P(s_i)$$

- Thus for $k = 3$ this becomes:

$$||[\neg\psi]||^3 = \neg P(s_0) \vee \neg P(s_1) \vee \neg P(s_2) \vee \neg P(s_3)$$

Final formula

- Thus the final formula $|[M]|^k \wedge |[\neg\psi]|^k$ for $k = 3$ becomes:

$$I(s_0) \wedge T(s_0, s_1) \wedge T(s_1, s_2) \wedge T(s_2, s_3) \wedge$$
$$(\neg P(s_0) \vee \neg P(s_1) \vee \neg P(s_2) \vee \neg P(s_3))$$

- If the formula is satisfiable, then an execution of the system of length 3 exists which violates the invariant property $P(s)$ in some state during the execution.

Reachability Diameter

- If the formula is unsatisfiable, we have proved that there is no execution of length at most 3 that violates the invariant.
- Clearly for every finite state system there is some bound d called the reachability diameter such that from the initial state every reachable state is reachable with an execution of at most length d.
- By taking $d = 2^n$, where n is the number of state bits, we could guarantee completeness.
- Unfortunately computing better approximations of d are computationally hard in the general case.

Unsatisfiable - Increase the bound

- Unfortunately the approach of taking $d = 2^n$ is not viable for anything but trivially small systems.
- Usually d is only increased by a small amount, say 1, and the procedure is repeated from the beginning until some resource limit (running time, memory, etc.) is hit.
- We will show a more refined approach to obtaining completeness later.

BMC: Pros and Cons

- Boolean formulas can be more compact than BDDs
- Leverages efficient SAT-solver technology
- Minimal length counterexamples (often, not always)
- Basic method is incomplete (we'll show some approaches to obtain completeness later in the tutorial)
- Not always better than BDD-based methods or explicit state model checking

Alternative Transition Relations

- When checking for reachability properties such as the violation of invariants, we can often replace the transition relation $T(s, s')$ with an alternative transition relation definition $T'(s, s')$ provided that:
  - Every state that is reachable from the initial state s_0 using $T(s, s')$ must be reachable from s_0 using $T'(s, s')$.
  - There should not be any new states reachable from s_0 using $T'(s, s')$ which are not reachable from s_0 using $T(s, s')$.

Encoding the Transition Relation

- There are now in fact many different ways to pick and encode an alternative transition relation $T'(s, s')$ if we consider asynchronous systems containing concurrency.
- A wish-list of mutually conflicting requirements for $T'(s, s')$ and its encoding:
  - Compact, hopefully linear in the size of the model.
  - Covers as many reachable states as possible for each bound k without losing soundness or completeness.
  - Efficiently solvable by the SAT solver.

Transition Relation Encoding

- Note that in the list of requirements we don't explicitly list that the number of state variables n should be minimised.
- This is often one of the main things to optimise with a BDD based symbolic model checker.
- Having too compact an encoding of the state vector can lead to losses in the SAT solver efficiency!
- More research is needed on how to more efficiently encode transition relations for different classes of systems. There are dramatic performance differences, at least for asynchronous systems.

Asynchronous Systems Case

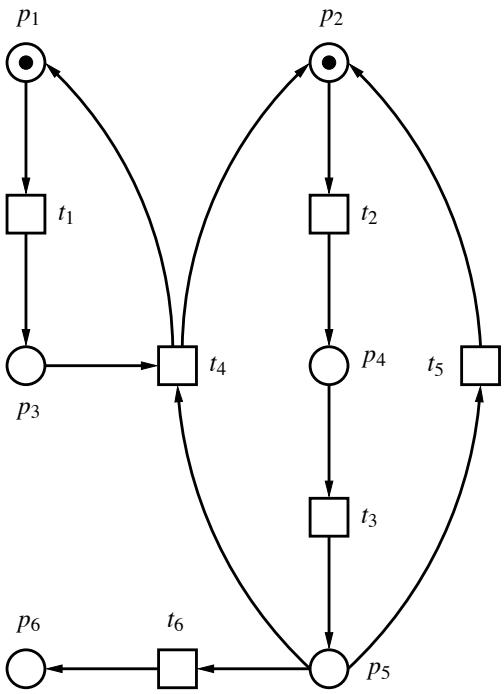
- We consider in this tutorial two simple classes of asynchronous systems but most of the results will carry over to more complicated models of concurrency.
- The two system models considered are:
  - 1-bounded Petri nets
  - Products of labelled transition systems (LTSs)

Petri nets

The class Petri nets we use are called place/transition nets (P/T-nets). A P/T-net is a tuple $N = (P, T, F, W, M_0)$, where

- P is a finite set of places,
- T is a finite set of transitions,
- $F \subseteq (P \times T) \cup (T \times P)$ is the flow relation,
- $W : F \mapsto \mathbb{N} \setminus \{0\}$ is the arc weight mapping, and
- $M_0 : P \mapsto \mathbb{N}$ is the initial marking.

Running Example P/T-net



The running Example

- Places $P = \{p_1, p_2, p_3, p_4, p_5, p_6\}$.
- Transitions $T = \{t_1, t_2, t_3, t_4, t_5, t_6\}$.
- Flow relation $F = \{(p_1, t_1), (t_1, p_3), (p_2, t_2), (t_2, p_4), (p_4, t_3), (t_3, p_5), (p_3, t_4), (p_5, t_4), (t_4, p_1), (t_4, p_2), (p_5, t_5), (t_5, p_2), (p_5, t_6), (t_6, p_6)\}$.
- Arc weight mapping $W(x, y) = 1$ for all $(x, y) \in F$. We use the convention that only arcs weights $W(x, y) > 1$ are drawn next to the arc (x, y), i.e., the default arc weight is 1.
- Initial marking $M_0 = \{p_1 \mapsto 1, p_2 \mapsto 1, p_3 \mapsto 0, p_4 \mapsto 0, p_5 \mapsto 0, p_6 \mapsto 0\}$.

Behaviour of P/T-nets

- The state of a P/T-net consist of a *marking* $M : P \mapsto \mathbb{N}$, which tells for each place how many *tokens* (drawn as black dots) it contains.
- The notation $M(p)$ denotes the number of tokens in place p.
- In our running example $M(p) \leq 1$ for all places $p \in P$, i.e., each place contains at most one token. However, this is not required in general.

Behaviour of P/T-nets

- The *preset* of a node $x \in P \cup T$ is denoted by ${}^\bullet x$ and defined to be: ${}^\bullet x = \{y \in P \cup T \mid (y,x) \in F\}$.
  The preset of a node consist of those nodes from which an arc to x exist. In our running example ${}^\bullet t_4 = \{p_3, p_5\}$.
- The *postset* of a node $x \in P \cup T$ is denoted by $x^\bullet$ and defined to be: $x^\bullet = \{y \in P \cup T \mid (x,y) \in F\}$.
  The postset of a node consist of those nodes to which an arc from x exist. In our running example $t_4{}^\bullet = \{p_1, p_2\}$.

Enabling of transitions

- A transition $t \in T$ is enabled in marking M, denoted $t \in enabled(M)$, if and only if (iff from now on) for all $p \in {}^\bullet t : M(p) \geq W(p,t)$.
  (All places p which are in the preset of t contain at least the number of tokens specified by $W(p,t)$.)

Firing of transitions

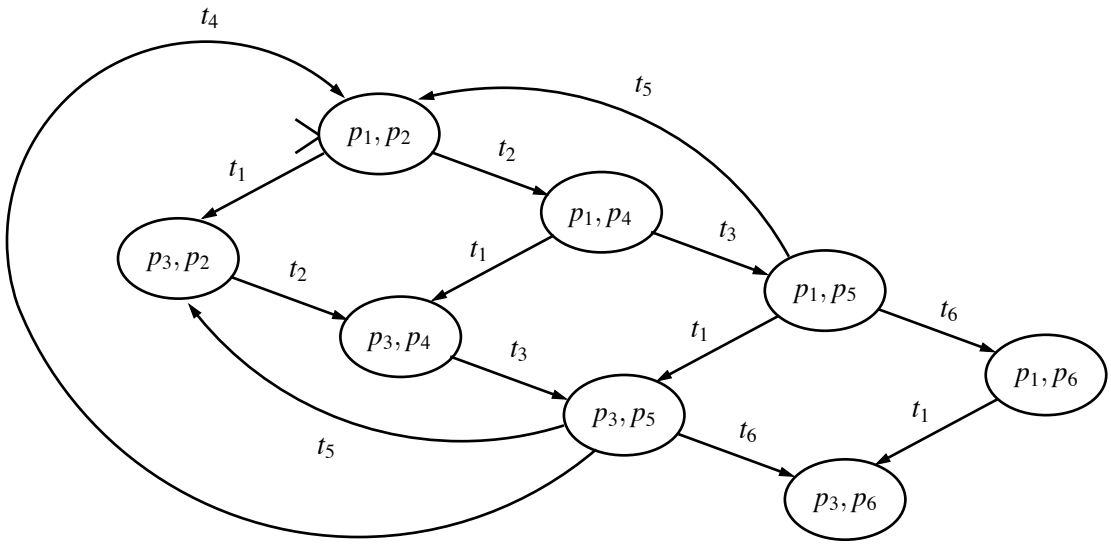
- To simplify definitions, we extend $W(x,y)$ to all pairs $(x,y) \in (P \cup T) \times (T \cup P)$ as follows: if $(x,y) \notin F$ then $W(x,y) = 0$.
- The marking M' reached after firing t, denoted $M' = \mathit{fire}(M,t)$, is defined for all $p \in P$ as: $M'(p) = M(p) - W(p,t) + W(t,p)$. (First remove as many tokens as given by $W(p,t)$ from all places in the preset of t, and then add as many tokens for all places in the postset of t as denoted by $W(t,p)$.)

Reachability graph

Reachability graph $G = (V, E, M_0)$ is the graph inductively defined as follows:

- $M_0 \in V$, where M_0 is the initial marking of the net N, and
- if $M \in V$ then for all $t \in \mathit{enabled}(M)$ it holds that $M' = \mathit{fire}(M,t) \in V$ and $(M,t,M') \in E$.

Reachability Graph



Reachability graph (cnt.)

- A place $p \in P$ is defined to be k-bounded iff for all reachable markings $M \in V$ it holds that $M(p) \leq k$.
- A net is defined to be k-bounded if all its places are k-bounded
- In the following we consider how to encode the transition relation $T(s, s')$ for 1-bounded P/T-nets only.

Interleaving Executions

- When the net is 1-bounded, we will use set notation for markings.
- In our running example the initial marking $M_0 = \{p_1, p_2\}$.
- In the initial marking the transition t_2 is enabled and its firing lead to marking $M' = \{p_1, p_4\}$. We denote this by: $\{p_1, p_2\}[t_2\rangle\{p_1, p_4\}$.
- One interleaving execution of length 4 leading to a deadlock; a marking with no enabled transitions is:

$$\{p_1, p_2\}[t_2\rangle\{p_1, p_4\}[t_3\rangle\{p_1, p_5\}[t_6\rangle\{p_1, p_6\}[t_1\rangle\{p_3, p_6\}$$

Conjunctive Normal Form (CNF)

- The SAT solvers mentioned so far require the problem to be mapped into the so called conjunctive normal form (CNF).
- A literal is either either a propositional variable x or its negation $\neg x$. A formula is in conjunctive normal form if it is a conjunction of clauses, where each clause is a disjunction of literals.
- Example: $(x \vee \neg y \vee \neg z) \wedge (\neg x \vee y) \wedge (\neg x \vee z)$ is in CNF.
- Using CNF formulas makes the implementation techniques inside SAT solvers simpler which often leads to more efficient implementation techniques.

Constrained Boolean Circuit SAT

- To raise the abstraction level a bit, in this tutorial we will use constrained Boolean circuit SAT instead of CNF.
- They are Boolean circuits where some of the output gates are constrained to *true*, while other can be constrained to *false*.
- The solver now has to find a valuation for the input variables of the circuit to make all the constraints to match the value computed by the circuit.

Boolean Circuit Format

- The format of the circuits used is described in http://www.tcs.hut.fi/~tjunttil/circuits/index.html. The page also contains constrained Boolean circuit front-ends to the zChaff and MiniSAT solvers, which internally convert the Boolean circuits into CNF.

From Circuits to CNF

- The page mentioned above also contains a tool called bc2cnf, with which you can obtain CNF formulas for other CNF based solvers.
- The translation to CNF is based on the Tseitin CNF encoding (see, e.g. page 562 of Towards an Efficient Tableau Method for Boolean Circuit Satisfiability Checking. Computational Logic 2000: 553-567, LNCS 1861.)
- Tseitin encoding introduces one new variable for each gate of the Boolean circuit, and then encodes the value of that gate with a small equivalence, translated into CNF.

From Circuits to CNF (cnt.)

- An AND gate $x := AND(y,z);$ would become: $x \Leftrightarrow (y \wedge z)$, and translated into CNF would give: $(x \vee \neg y \vee \neg z) \wedge (\neg x \vee y) \wedge (\neg x \vee z)$
- An OR gate $x := OR(y,z);$ would become: $x \Leftrightarrow (y \vee z)$, and translated into CNF would give: $(\neg x \vee y \vee z) \wedge (x \vee \neg y) \wedge (x \vee \neg z)$
- A constraint $x := true;$ will contribute to the CNF: x.
- A constraint $x := false;$ will contribute to the CNF: $\neg x$.

Cardinality Gates

- We also use a special gate type called the cardinality gate: $x := [0,1](a_0, a_1, \ldots, a_{m-1})$; which evaluates to true iff at most one of the m input variables $\{a_0, a_1, \ldots, a_{m-1}\}$ is true.
- This can be simulated with m new variables b_i, as follows: $b_0 := \mathit{false}$; for all $1 \leq i \leq m-1$ we have $b_i := b_{i-1} \vee a_{i-1}$; and the final value is obtained by $x := \neg((a_1 \wedge b_1) \vee \cdots \vee (a_{m-1} \wedge b_{m-1}))$;.
- There are also other linear size translations for replacing the cardinality gates with ANDs and ORs, the one above is picked for its simplicity.

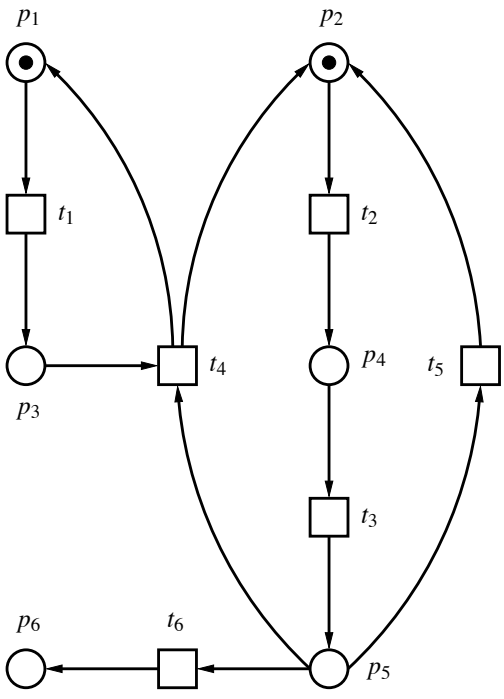
Cardinality Gates (cnt.)

- There is another $O(m^2)$ translation which does not introduce any new variables.
- It seems to have better performance for small values of m in CNF based SAT solvers.
- The use of cardinality gates is a vital ingredient to obtain a small encoding of the transition relation $T(s, s')$ for asynchronous systems.

The Transition Relation Encoding

- The following encoding almost identical to the one in: Keijo Heljanko: Bounded Reachability Checking with Process Semantics. CONCUR 2001: 218-232, LNCS 2154.

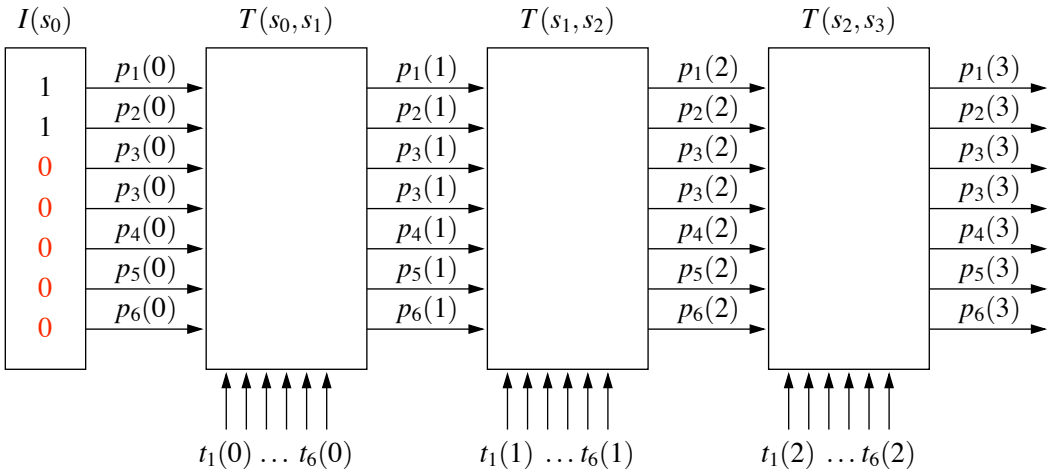
Running Example (recap)



State Variables and Inputs

- We now show how given a 1-bounded P/T-net its transition relation can be encoded into a constrained Boolean circuit.
- The mapping has the following intuition:
  - The state vector bit $p_j(i)$ $(= s_i[j])$ will be true iff in state s_i the place p_j contains a token. For example $p_3(0)$ is the variable corresponding to the place p_3 at the initial state s_0.
  - The Boolean circuit will have one free input variable $t_j(i)$ for each transition $t_j \in T$ and each transition relation instance $0 \leq i \leq k-1$. It $t_j(i)$ is true, then the transition t_j is fired at time i.

P/T-net Transition Relation Unrolling



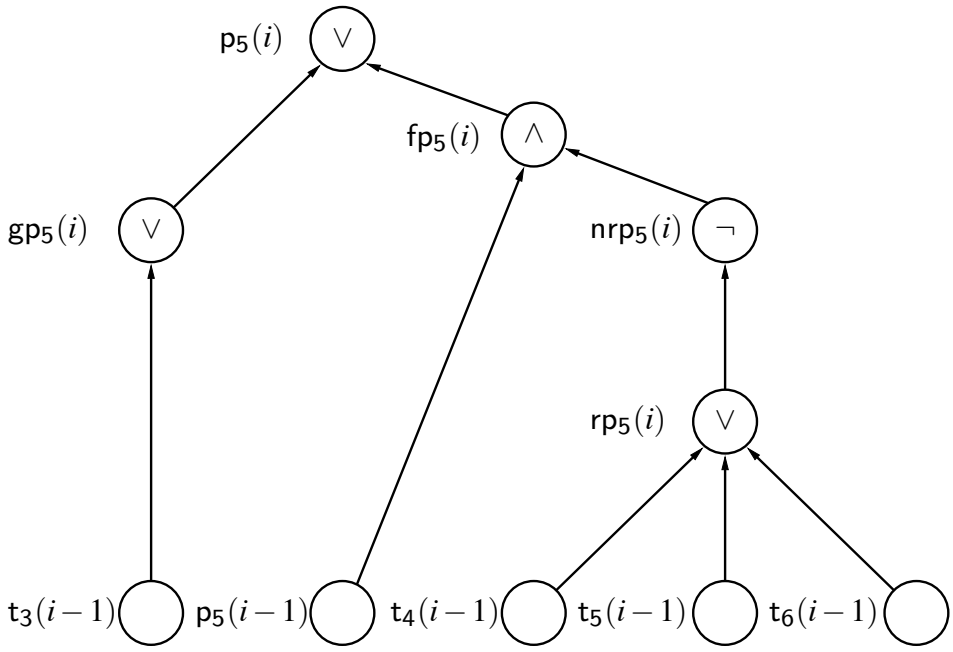
Initial Marking

- Handling the initial marking is easy, and goes as follows:
  - For each $p_j \in P$ such that $M(p) = 1$, set $p_j(0) := true$;
  - For each $p_j \in P$ such that $M(p) = 0$, set $p_j(0) := false$;

Token Updating

- For each place $p_j \in P$ and time $1 \leq i \leq k$ create the following gates:
  - $gp_j(i) := OR(t_1(i-1), t_2(i-1), \ldots, t_l(i-1));$, where ${}^\bullet p_j = \{t_1, t_2, \ldots, t_l\}$. This gate models the generation of a token to p_j.
  - $rp_j(i) := OR(t_1(i-1), t_2(i-1), \ldots, t_l(i-1));$, where $p_j{}^\bullet = \{t_1, t_2, \ldots, t_l\}$. This gate models the removal of a token from p_j.
  - $p_j(i) := gp_j(i) \vee (p_j(i-1) \wedge \neg rp_j(i));$. A token exists in p_j if either a new one was generated, or an old one existed and it was not removed.

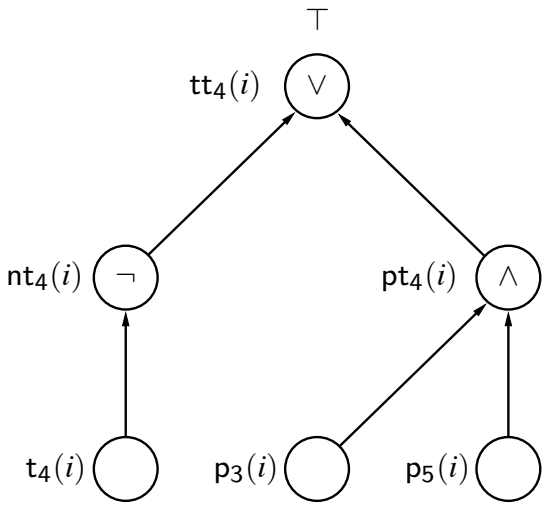
Translation for Place p_5



Transition Enabling

- We should also rule out models where a transition $t_j \in T$ is fired at time $0 \leq i \leq k-1$ without being enabled by using the following gates:
  - Create a gate $pt_j(i) := AND(p_1(i), p_2(i), \ldots, p_l(i))$;, where ${}^\bullet t_j = \{p_1, p_2, \ldots, p_l\}$. This gate is true when the transition t_j is enabled.
  - Disallow the firing of disabled transitions with a gate: $tt_j(i) := \neg t_j(i) \vee pt_j(i)$, where $tt_j(i)$ is constrained to *true*. (This is simply a constrained Boolean circuit encoding of $t_j(i) \Rightarrow pt_j(i)$.)

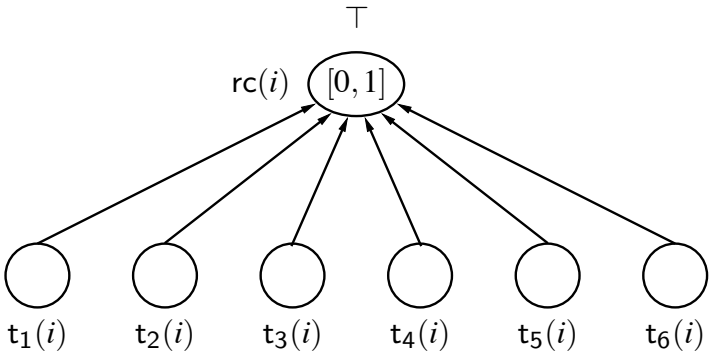
Translation for Transition t_4



Removing Conflicting Transitions

- The encoding so far allows for several transitions to be fired concurrently even if they are in conflict.
- We will disallow this by adding the following gate at each time point $0 \leq i \leq k-1$:
  - $rc(i) := [0,1](t_1(i), t_2(i), \ldots, t_l(i))$, where $T = \{t_1, t_2, \ldots, t_l\}$. We also constrain the gate $rc(i)$ to *true*.
  - The gate $rc(i)$ intuitively removes the possibility of two conflicting transitions to fire at the same time point because at most one transition is allowed to fire at each time point.

Removing Conflicts



Interleaving Semantics

- All parts of the encoding taken together give a constrained Boolean circuit encoding of $T(s, s')$ for the interleaving semantics.
- The interleaving semantics is the standard textbook semantics of P/T-nets where at most one transition is allowed to fire at each time point.
- The size of the encoding is linear in both the size of the input P/T-net and the bound k.

Optional Idling Removal

- The circuit as show above allows no transition to be fired at any index i of the execution.
- It is easy to disallow this by for all $0 \leq i \leq k-1$ introducing a gate: $idle(i) := \neg(OR(t_1(i), t_2(i), \ldots, t_l(i)))$;, where $T = \{t_1, t_2, \ldots, t_l\}$. We can now optionally constrain the gate $idle(i)$ to $false$ to remove idling.
- If idling is not removed, the encoding has models corresponding to all executions of length k or less.
- When idling is removed, the encoding has models corresponding to all executions of exactly length k.

Deadlock Detection

- We now assume idling has not been removed, and also drop the assumption that $T(s, s')$ is total.
- Deadlocking executions of length k or less can now be captured by adding the following constraint on the places $p_i(k)$:
- First add the translation of the transition preset gates $pt_j(i)$ also for the index $i = k$.
- Add a gate $dead(k) := \neg(OR(pt_1(k), pt_2(k), \ldots, pt_l(k)))$;, where $T = \{t_1, t_2, \ldots, t_l\}$. Constrain the gate $dead(k)$ to $true$ to capture executions leading to a state where no transition is enabled: a deadlock state.

Deadlock Checking Demo (1/3)

```
$ cat running.net
P = ['p1','p2','p3','p4','p5','p6']
T = ['t1','t2','t3','t4','t5','t6']
F = [['p1','t1'],['t1','p3'],
['p2','t2'],['t2','p4'],
['p4','t3'],['t3','p5'],
['p3','t4'],['p5','t4'],['t4','p1'],['t4','p2'],
['p5','t5'],['t5','p2'],
['p5','t6'],['t6','p6']]
M_0 = ['p1','p2']
bound = 4
semantics = "interleaving"

$ 1b-pn-bmc < running.net | bczchaff | cex-print
{p1, p2}[t1>{p2, p3}[t2>{p3, p4}[t3>{p3, p5}[t6>{p3, p6}
```

Deadlock Checking Demo (2/3)

```
$ 1b-pn-bmc < running.net | bczchaff -v
Parsing from stdin
The circuit has 196 gates
The input gates are: t6_3 t3_3 t2_3 t5_3 t1_3 t4_3 t6_2 t3_2 t2_2 t5_2 t1_2 t4_2 t6_1 t3_1 t2_1 t5_1
t1_1 t4_1 t6_0 t3_0 t2_0 t5_0 t1_0 t4_0
The circuit has 138 gates and 153 edges after simplification
The circuit has 83 gates and 134 edges after sharing
The circuit has 75 gates and 92 edges after simplification
The circuit has 59 gates and 89 edges after sharing
The circuit has 54 gates and 66 edges after simplification
The circuit has 48 gates and 65 edges after sharing
The circuit has 48 gates and 65 edges after simplification
The circuit has 48 gates and 65 edges after sharing
The circuit has 56 gates after normalization
The circuit has 56 gates and 71 edges after simplification
The circuit has 52 gates and 71 edges after sharing
The circuit has 52 gates and 71 edges after simplification
The circuit has 52 gates and 71 edges after sharing
The max-min height of the circuit is 2
The max-max height of the circuit is 4
The circuit has 46 relevant gates
The circuit has 10 relevant input gates
The cnf has 37 variables and 96 clauses
```

Deadlock Checking Demo (3/3)

Executing zchaff...

Max Decision Level 1
Num. of Decisions 2
Original Num Clauses 96
Original Num Literals 200
Added Conflict Clauses 0
Added Conflict Literals 0
Deleted Unrelevant clause 0
Deleted Unrelevant literals 0
Number of Implication 37
~live_4 ~gp4_4 ~t2_3 ~gp5_4 ~t3_3 ~gp1_4 ~t4_3 ~t5_3 ~gp1_2 ~t4_1 ~t5_1 ~t5_0 ~gp1_1
~t4_0 ~p5_4 ~p4_4 ~p2_4 ~p1_4 ~p6_2 ~gp6_2 ~t6_1 ~p6_1 ~gp6_1 ~t6_0 ~p5_1 ~gp5_1
~t3_0 ~p6_0 ~p5_0 ~p4_0 ~p3_0 ~gp1_3 ~t4_2 ~t5_2 ~gp2_1 ~gp2_2 ~gp2_4 ~gp4_3 ~t2_2
~p4_3 ~p2_3 ~gp2_3 rc1 rc2 gate13 gate15 gate16 gate11 gate10 gate9 rc0 gate5 gate4 gate3
gate2 gate1 gate0 p2_0 p1_0 gate19 gate20 gate21 gate22 rc3 gate23 gate18 p6_4 gp6_4 t6_3
p3_4 ~gp3_4 ~t1_3 gate17 gate14 gate12 p5_3 ~p6_3 ~gp6_3 ~t6_2 gp5_3 t3_2 p3_3 ~p1_3
~gp3_3 ~t1_2 gate8 gate7 gate6 p4_2 ~p5_2 ~gp5_2 ~t3_1 p3_2 ~p2_2 gp4_2 t2_1 ~p1_2
~gp3_2 ~t1_1 p2_1 ~p4_1 ~gp4_1 ~t2_0 ~p1_1 p3_1 gp3_1 t1_0
Satisfiable

Step Semantics

- An old well known semantics from the theory of Petri nets is the so called step semantics.
- This is not the same as maximal step semantics.
- The idea is the following: Instead of firing a single enabled transition at each marking M, we can fire a step: a set of enabled transitions $S \subseteq enabled(M)$ at a time point provided they are all pairwise concurrent:
  - For all pairs of distinct transitions $t, t' \in S$ it holds that ${}^\bullet t \cap {}^\bullet t' = \emptyset$.
  - Note: It is straightforward to prove that because we only consider 1-bounded P/T-nets here that actually also $({}^\bullet t \cup t^\bullet) \cap ({}^\bullet t' \cup t'^\bullet) = \emptyset$ holds.

Step Reachability graph

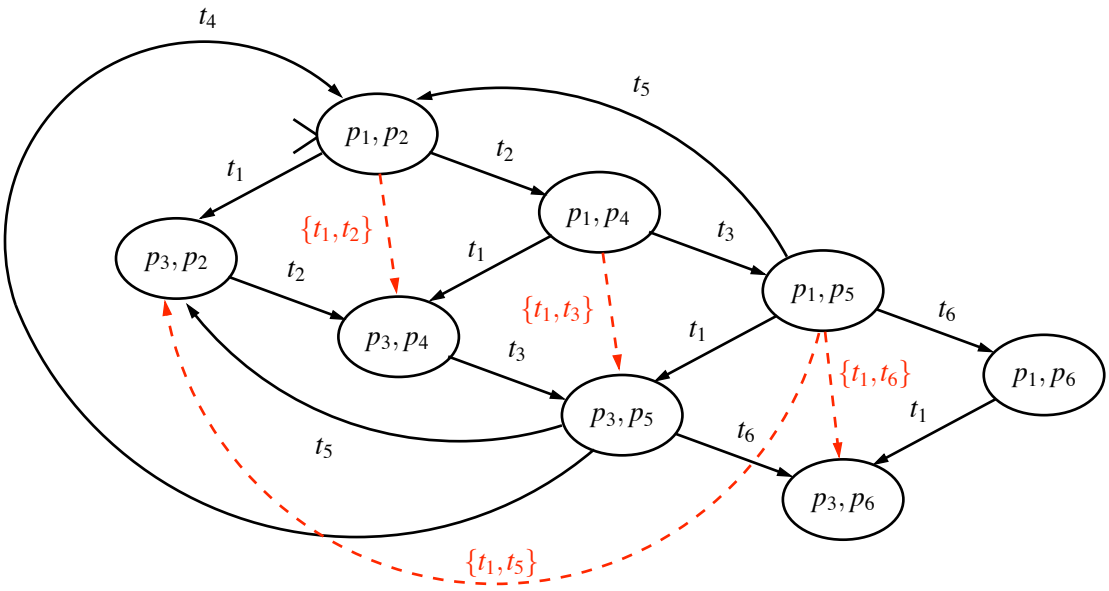
Step reachability graph $G_s = (V, E, M_0)$ is the graph inductively defined as follows:

- $M_0 \in V$, where M_0 is the initial marking of the net N, and
- if $M \in V$ then for all $S \subseteq enabled(M)$ such that S is a step it holds that $M' = fire(M, S) \in V$ and $(M, S, M') \in E$.

Some Properties of Steps

- If a set of transitions $S = \{t_1, t_2, \ldots, t_l\}$ is step enabled in M, then all the $l!$ interleaving executions obtained by sequentialising S in all orders are enabled interleaving executions in M, and they all lead to the same final state.
- An intuition why this is the case: Because $({}^\bullet t \cup t^\bullet) \cap ({}^\bullet t' \cup t'^\bullet) = \emptyset$, the transitions t and t' happen "in different parts of the system", and thus cannot influence each other in any way.
- Thus $fire(M, S)$ from the previous slide can be defined as: $fire(\ldots(fire(fire(M, t_1), t_2), \ldots, t_l)$.

Step Reachability Graph



Properties Steps Graphs

- Because all singleton sets are also steps, the (interleaving) reachability graph is always a subgraph of the step reachability graph.
- Because the final state reached after firing a step is the final state of every interleaving of the step, no new reachable states have been introduced.
- The reachability diameter of the system is in the worst case as big as in the interleaving case.
- In the best case the interleaving diameter has become smaller, because a step with o transitions has to be simulated with o time steps in the interleaving reachability graph.

Running Example: Steps

- We extend the notation $M[t\rangle M'$ to also denote the firing of steps $M[S\rangle M'$.
- In the running example one step executions of length 3 leading to the deadlock marking $\{p_3, p_6\}$ is:

$$\{p_1, p_2\}[t_2\rangle\{p_1, p_4\}[t_1, t_3\rangle\{p_3, p_5\}[t_6\rangle\{p_3, p_6\}$$

- Recall that the shortest execution to $\{p_3, p_6\}$ was of length 4 in the interleaving reachability graph.
- $\Rightarrow$ Using step semantics allows one to sometimes detect errors with smaller bounds.

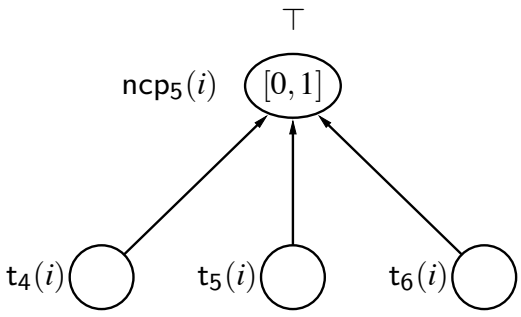
Encoding the Step Semantics

- It is easy to modify the interleaving transition relation encoding to encode step semantics instead.
- The only thing we have to remove is the encoding of gate $rc(i)$, which restricts the number of fired transitions to at most one as required by the interleaving case.
- A set of constraints has to be added to remove the possibility of two conflicting transitions to fire in the same step.

Steps: Removing Conflicts

- For each place $p_j \in P$ and each step $0 \leq i \leq k-1$ we will add the following gate which disallows concurrent firing of transitions which are not concurrent due to both having the place p_j in their preset:
- $ncp_j(i) := [0,1](t_1(i), t_2(i), \ldots, t_l(i))$, where $p_j{}^\bullet = \{t_1, t_2, \ldots, t_l\}$. We also constrain the gate $ncp_j(i)$ to *true*.

Steps: Removing Conflicts wrt. p_5



Demo with Step Semantics (1/3)

```
$ cat step-running.net
P = ['p1','p2','p3','p4','p5','p6']
T = ['t1','t2','t3','t4','t5','t6']
F = [['p1','t1'],['t1','p3'],
['p2','t2'],['t2','p4'],
['p4','t3'],['t3','p5'],
['p3','t4'],['p5','t4'],['t4','p1'],['t4','p2'],
['p5','t5'],['t5','p2'],
['p5','t6'],['t6','p6']]
M_0 = ['p1','p2']
bound = 3
semantics = "step"

$ 1b-pn-bmc < step-running.net | bczchaff | cex-print
{p1, p2}[t2>{p1, p4}[t1, t3>{p3, p5}[t6>{p3, p6}
```

Demo with Step Semantics (2/3)

```
$ 1b-pn-bmc < step-running.net | bczchaff -v
Parsing from stdin
The circuit has 149 gates
The input gates are: t6_2 t3_2 t2_2 t5_2 t1_2 t4_2 t6_1 t3_1 t2_1 t5_1 t1_1 t4_1 t6_0 t3_0 t2_0 t5_0 t1_0 t4_0
The circuit has 99 gates and 92 edges after simplification
The circuit has 48 gates and 74 edges after sharing
The circuit has 39 gates and 33 edges after simplification
The circuit has 27 gates and 32 edges after sharing
The circuit has 24 gates and 16 edges after simplification
The circuit has 14 gates and 16 edges after sharing
The circuit has 14 gates and 16 edges after simplification
The circuit has 14 gates and 16 edges after sharing
The circuit has 14 gates after normalization
The circuit has 14 gates and 16 edges after simplification
The circuit has 14 gates and 16 edges after sharing
The max-min height of the circuit is 2
The max-max height of the circuit is 3
The circuit has 10 relevant gates
The circuit has 3 relevant input gates
The cnf has 8 variables and 15 clauses
```

Demo with Step Semantics (3/3)

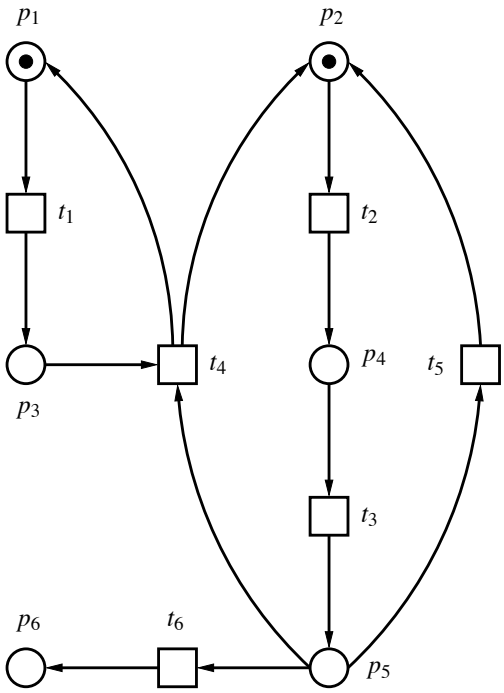
Executing zchaff...

Max Decision Level 2
Num. of Decisions 3
Original Num Clauses 15
Original Num Literals 31
Added Conflict Clauses 0
Added Conflict Literals 0
Deleted Unrelevant clause 0
Deleted Unrelevant literals 0
Number of Implication 8
~p2_1 ~p2_2 ~p4_2 ~t2_1 ~gp4_2 ~gp2_3 ~gp2_2 ~gp2_1 ~p3_0 ~p4_0 ~p5_0 ~p6_0 ~t3_0 ~gp5_1 ~p5_1 ~t6_0
~gp6_1 ~p6_1 ~t6_1 ~gp6_2 ~p6_2 ~p1_3 ~p2_3 ~p4_3 ~p5_3 ~t4_0 ~gp1_1 ~t5_0 ~t5_1 ~t2_2 ~gp4_3 ~t3_2
~gp5_3~t4_1 ~gp1_2 ~t4_2 ~gp1_3 ~t5_2 ~live_3 t2_0 gp4_1 p4_1 t3_1 gp5_2 p5_2 gate7 gate8 t6_2
gp6_3 p6_3 gate16 gate15 gate14 gate13 ncp5_2 p1_0 p2_0 gate0 gate1 gate2 gate3 gate4 gate5
gate9 gate10 gate11 ncp5_0ncp5_1 gate17 gate12 p3_3 ~gp3_3 ~t1_2 gate6 p3_2 ~p1_2 gp3_2 t1_1
p1_1 ~p3_1 ~gp3_1 ~t1_0
Satisfiable

Interleaving vs. Steps

- We have not yet found a domain where the interleaving encoding would be superior in performance to the step encoding.
- Quite often even small reductions in the required bound translate to large performance differences.
- The step encoding also is more "local" than the interleaving encoding:
  - Parts of the system which do not share resources are never linked together as done by the $rc(i)$ gate in the interleaving case.
  - This might have SAT performance implications.

Running Example (recap2)



Process Semantics

- In our running example there are three step executions of length 3 leading to the deadlock marking $\{p_3, p_6\}$:

$$\{p_1, p_2\}[t_2\rangle\{p_1, p_4\}[t_3\rangle\{p_1, p_5\}[t_1, t_6\rangle\{p_3, p_6\}$$

$$\{p_1, p_2\}[t_2\rangle\{p_1, p_4\}[t_1, t_3\rangle\{p_3, p_5\}[t_6\rangle\{p_3, p_6\}$$

$$\{p_1, p_2\}[t_1, t_2\rangle\{p_3, p_4\}[t_3\rangle\{p_3, p_5\}[t_6\rangle\{p_3, p_6\}$$

- Intuitively they all correspond a concurrent execution where "the component on the left" executes t_1, and "the component on the right" executes the sequence t_2, t_3, t_6.

Process Semantics (cnt.)

- Can we somehow pick a unique canonical representative of such “concurrent” behaviour, and thus reduce the number of different executions the SAT solver has to consider?
- The answer turns out to be positive. The resulting semantics will be called process semantics.
- There is even a compact SAT encoding to capture the process semantics!

The Process Normal Form

- A step execution $M_0[S_0\rangle M_1[S_1\rangle \cdots M_{k-1}[S_{k-1}\rangle M_k$ is in process normal form iff for every index $i \geq 1$ and every transition $t_j \in S_i$ it holds that:
  - There is some transition $t' \in S_{i-1}$ such that $t'^{\bullet} \cap {}^{\bullet}t_j \neq \emptyset$.

Process Normal Form Intuition

- Intuitively the above means: In a step execution in process normal form each transition is executed at the earliest time moment all its tokens are available.
- In the SAT encoding setting this means that a transition should be enabled only if one of its tokens has been generated in the previous step.
- Thus the process execution for the example is:

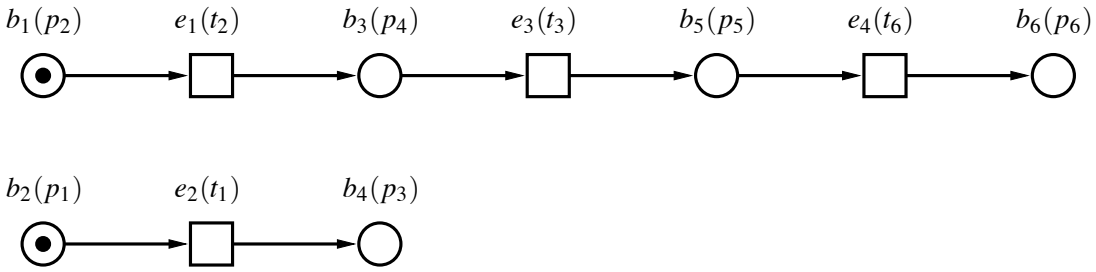
$$\{p_1, p_2\}[t_1, t_2\rangle\{p_3, p_4\}[t_3\rangle\{p_3, p_5\}[t_6\rangle\{p_3, p_6\}$$

Normalising a Step Execution

- By repeatedly running the following simple algorithm each step execution $M_0[S_0\rangle M_1[S_1\rangle \cdots M_{k-1}[S_{k-1}\rangle M_k$ can be converted into a process executions of at most the same length and leading to the same final state:
  - Take a transition $t_j \in S_i$ which violates the process condition.
  - Remove t_j from S_i and add it to S_{i-1}.
- Proof is simple, one has to show that: (i) t_j is enabled already in S_{i-1}, and (ii) S_{i-1} contains no transitions in conflict with t_j.

Process

As a graphical presentation of the process we can again use a P/T-net:



The step executions in process normal form correspond to slicing the net one level at a time starting from the left.

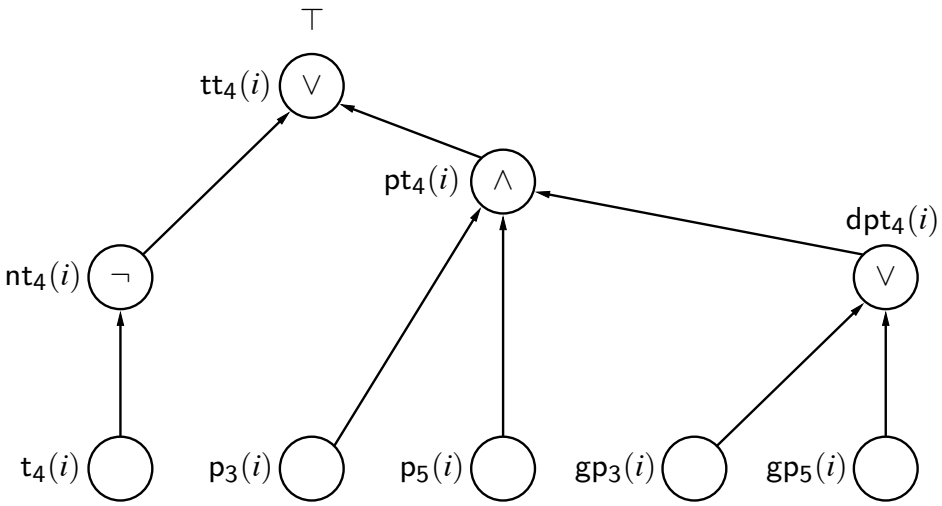
Properties of Processes

- Each state of the system is reachable by a process execution that is among the shortest step executions to reach that state.
- Thus the set of reachable states is preserved.
- Furthermore, the process reachability diameter is always as small as the step reachability diameter.
- There are at most as many process executions of length k as there are interleaving executions of length k.
- There can be exponentially more step and interleaving executions of length k than there are process executions.

Encoding Process Semantics

- Take the encoding for the step semantics but change the transition enabling gate definition for all $i > 0$:
- For each transition $t_j \in T$ and time $1 \leq i \leq k-1$ use the following gates (for $i = 0$ use the step version):
  - Create a gate $pt_j(i) := AND(p_1(i), p_2(i), \ldots, p_l(i), OR(gp_1(i), gp_2(i), \ldots, gp_l(i)))$;, where ${}^\bullet t_j = \{p_1, p_2, \ldots, p_l\}$. This gate is true when the transition t_j is enabled, and at least one of the tokens has been freshly generated.
  - Add gate: $tt_j(i) := \neg t_j(i) \vee pt_j(i)$, and constrain it to $true$.

Process Translation for t_4



Demo with Process Semantics (1/3)

```
$ cat process-running.net
P = ['p1','p2','p3','p4','p5','p6']
T = ['t1','t2','t3','t4','t5','t6']
F = [['p1','t1'],['t1','p3'],
['p2','t2'],['t2','p4'],
['p4','t3'],['t3','p5'],
['p3','t4'],['p5','t4'],['t4','p1'],['t4','p2'],
['p5','t5'],['t5','p2'],
['p5','t6'],['t6','p6']]
M_0 = ['p1','p2']
bound = 3
semantics = "process"

$ 1b-pn-bmc < process-running.net | bczchaff | ./cex-print
{p1, p2}[t1, t2>{p3, p4}[t3>{p3, p5}[t6>{p3, p6}
```

Demo with Process Semantics (2/3)

```
$ 1b-pn-bmc < process-running.net | bczchaff -v
Parsing from stdin
The circuit has 161 gates
The input gates are: t6_2 t3_2 t2_2 t5_2 t1_2 t4_2 t6_1 t3_1 t2_1 t5_1 t1_1 t4_1 t6_0 t3_0 t2_0 t5_0 t1_0 t4_0
The circuit has 101 gates and 88 edges after simplification
The circuit has 44 gates and 70 edges after sharing
The circuit has 30 gates and 11 edges after simplification
The circuit has 10 gates and 10 edges after sharing
The circuit has 8 gates and 0 edges after simplification
The circuit has 2 gates and 0 edges after sharing
The circuit has 2 gates and 0 edges after simplification
The circuit has 2 gates and 0 edges after sharing
The circuit has 2 gates after normalization
The circuit has 2 gates and 0 edges after simplification
The circuit has 2 gates and 0 edges after sharing
The max-min height of the circuit is 0
The max-max height of the circuit is 0
The circuit has 0 relevant gates
```

Note that with the more constrained process encoding, the preprocessing already solves the circuit.

Demo with Process Semantics (3/3)

```
~p1_1 ~p2_1 ~p2_2 ~p1_2 ~gp2_2 ~gp2_1 ~gp2_3 ~t2_1 ~gp4_2 ~p4_2 ~p3_0 ~p4_0 ~p5_0 ~p6_0 ~t3_0
~gp5_1 ~p5_1 ~t6_0 ~gp6_1 ~p6_1 ~t6_1 ~gp6_2 ~p6_2 ~t1_2 ~gp3_3 ~p1_3 ~p2_3 ~p4_3 ~p5_3 ~t5_0
~t4_0 ~gp1_1 ~t5_1 ~t2_2 ~gp4_3 ~t3_2 ~gp5_3 ~t4_1 ~gp1_2 ~t1_1 ~gp3_2 ~t4_2 ~gp1_3 ~t5_2 ~live_3
t3_1 gp5_2 p5_2 gate8 t6_2 gp6_3 p6_3 gate16 p3_3 p3_2 gate15 gate14 gate13 p1_0 p2_0 gate0
gate1 gate2 gate3 gate4 gate5 gate6 gate9 gate10 gate11 gate12 ncp5_2 p4_1 gp4_1 t2_0 gate7
ncp5_0 ncp5_1 p3_1 gp3_1 t1_0 gate17
Satisfiable
```

No need to invoke zChaff, just output the solution.

Steps vs. Processes

- Unfortunately there is some bad news: processes are not always faster than steps with the latest SAT solvers such as zChaff and Siege. (Cause unknown.)
- Often a polynomial time preprocessing algorithm is used to compute the earliest time each transition can fire or a place can become marked.
- This allows for simplification of the BMC encoding by introducing constant values for variables.
- Process semantics can be used as a better polynomial time preprocessing step as it can prove for more transitions that they can never be enabled at certain time points in process executions.

Steps vs. Processes (cnt.)

- The transition relation for steps can be represented as: $T(s_i, i_i, s_{i+1})$, where i_i is the current input vector (in the running example, the set of transitions to be fired in step S_i: $i_i = \langle t_1(i), t_2(i), \ldots, t_6(i) \rangle$).
- The transition relation for processes can be represented as: $T(s_i, i_{i-1}, i_i, s_{i+1})$, where i_i is the current input vector (step S_i) and i_{i-1} is the previous input vector (step S_{i-1}).

History Dependence

- Thus the process semantics semantics has a history dependent transition relation.
- Another way to present this is to use "three valued tokens", a place can either contain: no tokens, contain a freshly generated token, or contain an old token.
- This makes the use of process semantics unattractive in a BDD model checking setting, as the number of state bits needed to represent the state vector grows.

Processes and Temporal Induction

- It follows from Theorem 17 in Toni Jussila's Doctoral dissertation that the *SimplePath* constraint used in the temporal induction (k-induction) method (to be presented later in this tutorial) can treat old tokens and fresh tokens alike.
- Thus for reachability from the initial state the so called recurrence diameter (to be defined later) for processes is never worse for processes than for steps, but can sometimes even be better.

Model Checking LTL-X

- One can also do model checking of the temporal logic LTL-X with step semantics. (Extension to allow also processes to be used is work in progress.)
- LTL-X is the subset of LTL where the next-time operator X has been removed. This restriction of the logic is often done also with other partial order methods.
- First one has to identify all visible transitions of the net, which can modify the truth value of some atomic proposition in the formula.

Model Checking LTL-X (cnt.)

- All of the visible transitions are made to conflict with each other by adding a new marked place v to the net, and adding a bidirectional arc from each visible transition to v.
- If the is deadlock free, one can additionally require that the last step S_k is non-empty to disallow illegal counterexamples by infinite sequences of idling.
- If the net can deadlock, the solution is more subtle, and we refer to our paper on the subject:
  Keijo Heljanko, Ilkka Niemelä:
  Bounded LTL model checking with stable models.
  TPLP 3(4-5): 519-550 (2003), Cambridge University Press.

Conclusions of Tutorial part 1

- Bounded model checking (BMC) is an efficient way of implementing *symbolic model checking*.
- It alleviates the state explosion by representing the state space implicitly as a propositional formula.
- It leverages efficient SAT-solver technology.
- The choice between different transition relation encodings has been often overlooked in BMC literature.
- The performance differences between different transition relation encodings are very significant, at least for asynchronous systems BMC.

Chapter 8 : Specification Languages and Formal Description techniques

Formal methods

Example of formal methods (from Wikipedia)

- Abstract State Machines (ASMs)
- Alloy
- B-Method
- Process calculi or process algebrae
  - CCS
  - CSP
  - LOTOS
  - π-calculus
- Actor model
- Esterel
- Lustre
- Petri nets
- RAISE
- VDM
- VDM-SL
- VDM++
- Z notation

Plan

Process calculi (from wikipedia)

A process calculus or process algebra

- provide a tool for the high-level description of interactions, communications, and synchronizations between a collection of independent agents or processes
- models open or closed systems
- provide algebraic laws that allow process descriptions to be manipulated and analyzed,
- permit formal reasoning about equivalences between processes (e.g., using bisimulation, failure-divergence equivalence, ...).

Essential features of process algebra

Essential features of process algebra

- communication via synchronization or message-passing rather than as the modification of shared variables
- Describing processes and systems using a small collection of primitives, and operators for combining those primitives
- Defining algebraic laws for the process operators, which allow process expressions to be manipulated using equational reasoning

Basic concepts in a process algebra

Basic concepts in a process algebra

- Events :
  - internal to the process it belongs to (often denoted τ, ϵ or i)
  - external : will take place in a synchronization with one or several other process(es).
- operators : e.g.
  - parallel composition of processes
  - sequentialization of interactions
  - choice
  - hiding of interaction points
  - recursion or process replication

CSP

CSP

- Formal description language together with a formal method
- Process algebra
- CSP initial semantics is called Failures-Divergences model
- CSP has also an operational semantics

History

- First presented in Hoare's original 1978 paper
- Hoare, Stephen Brookes, and A. W. Roscoe developed and refined the theory of CSP into its modern, process algebraic form.
- The theoretical version of CSP was initially presented in a 1984 article by Brookes, Hoare, and Roscoe, and later in Hoare's book Communicating Sequential Processes, which was published in 1985.

CSP

Note

This section is taken from Jeremy Martin's PhD thesis

CSP

Syntax

$$
\begin{array}{rcl}
Process & :== & STOP \mid \\
 & & SKIP \mid \\
 & & event \rightarrow Process \mid \\
 & & Process \,;\, Process \mid \\
 & & Process \;[\![alph \mid alph]\!]\; Process \mid \\
 & & Process \;|||\; Process \mid \\
 & & Process \sqcap Process \mid \\
 & & Process \;\Box\; Process \mid \\
 & & Process \setminus event \mid \\
 & & f(Process) \mid \\
 & & name \mid \\
 & & \mu\, name \bullet Process
\end{array}
$$

CSP

A vending machine example

- The system is the synchronization of the vending machine VM and the Tea Drinker TD

$$VM = coin \rightarrow ((tea \rightarrow VM) \;\Box\; (coin \rightarrow coffee \rightarrow VM))$$

$$TD = (coin \rightarrow tea \rightarrow TD) \;\Box\; (coffee \rightarrow TD)$$

$$
\begin{array}{rcl}
SYSTEM & = & VM \;[\![\{coin,coffee,tea\} \mid \{coin,coffee,tea\}]\!]\; TD \\
 & = & \left(\begin{array}{c} (coin \rightarrow ((tea \rightarrow VM) \;\Box\; (coin \rightarrow coffee \rightarrow VM))) \\ [\![\{coin,coffee,tea\} \mid \{coin,coffee,tea\}]\!] \\ ((coin \rightarrow tea \rightarrow TD) \;\Box\; (coffee \rightarrow TD)) \end{array} \right) \\
 & = & coin \rightarrow \left(\begin{array}{c} ((tea \rightarrow VM) \;\Box\; (coin \rightarrow coffee \rightarrow VM)) \\ [\![\{coin,coffee,tea\} \mid \{coin,coffee,tea\}]\!] \\ tea \rightarrow TD \end{array} \right) \\
 & & \text{using law 1.22 with } X = \{coin\},\, Y = \{coin,coffee\},\, Z = \{coin\} \\
 & = & coin \rightarrow tea \rightarrow (VM \;[\![\{coin,coffee,tea\} \mid \{coin,coffee,tea\}]\!]\; TD) \\
 & & \text{using law 1.22 with } X = \{tea,coin\},\, Y = \{tea\},\, Z = \{tea\} \\
 & = & coin \rightarrow tea \rightarrow SYSTEM
\end{array}
$$

CSP

Axiomatic laws

$$SKIP\,;P = P\,;SKIP = P \tag{1.1}$$
$$STOP\,;P = STOP \tag{1.2}$$
$$(P\,;Q)\,;R = P\,;(Q\,;R) \tag{1.3}$$
$$(a \to P)\,;Q = a \to (P\,;Q) \tag{1.4}$$
$$P \;[\![\, A \,|\, B \,]\!]\; Q = Q \;[\![\, B \,|\, A \,]\!]\; P \tag{1.5}$$
$$P \;[\![\, A \,|\, B \cup C \,]\!]\; (Q \;[\![\, B \,|\, C \,]\!]\; R) = (P \;[\![\, A \,|\, B \,]\!]\; Q) \;[\![\, A \cup B \,|\, C \,]\!]\; R \tag{1.6}$$
$$P \;|||\; Q = Q \;|||\; P \tag{1.7}$$
$$P \;|||\; SKIP = P \tag{1.8}$$
$$P \;|||\; (Q \;|||\; R) = (P \;|||\; Q) \;|||\; R \tag{1.9}$$
$$P \sqcap P = P \tag{1.10}$$
$$P \sqcap Q = Q \sqcap P \tag{1.11}$$
$$P \sqcap (Q \sqcap R) = (P \sqcap Q) \sqcap R \tag{1.12}$$

CSP

Axiomatic laws (cont'd)

$$P \,\Box\, P = P \tag{1.13}$$
$$P \,\Box\, Q = Q \,\Box\, P \tag{1.14}$$
$$P \,\Box\, (Q \,\Box\, R) = (P \,\Box\, Q) \,\Box\, R \tag{1.15}$$
$$P \;[\![\, A \,|\, B \,]\!]\; (Q \sqcap R) = (P \;[\![\, A \,|\, B \,]\!]\; Q) \sqcap (P \;[\![\, A \,|\, B \,]\!]\; R) \tag{1.16}$$
$$P \,\Box\, (Q \sqcap R) = (P \,\Box\, Q) \sqcap (P \,\Box\, R) \tag{1.17}$$
$$P \sqcap (Q \,\Box\, R) = (P \sqcap Q) \,\Box\, (P \sqcap R) \tag{1.18}$$
$$(x \to P) \,\Box\, (x \to Q) = (x \to P) \sqcap (x \to Q) = x \to (P \sqcap Q) \tag{1.19}$$
$$P \,\Box\, STOP = P \tag{1.20}$$
$$\Box_{x:\{\}}\; x \to P_x = STOP \tag{1.21}$$

CSP

Axiomatic laws (cont'd)

$$
\begin{array}{rrcll}
\text{Let} & P & = & \Box_{x:X}\, x \to P_x & \\
 & Q & = & \Box_{y:Y}\, y \to Q_y & \\
\text{Then} & P \;[\![A \,|\, B]\!]\; Q & = & \Box_{z:Z}\, z \to (P_z' \;[\![A \,|\, B]\!]\; Q_z') & \\
\text{where} & P_z' & = & \{ \begin{array}{ll} P_z & \textit{if} \quad z \in X \\ P & \text{otherwise} \end{array} & \\
\text{and} & Q_z' & = & \{ \begin{array}{ll} Q_z & \textit{if} \quad z \in Y \\ Q & \text{otherwise} \end{array} & \\
\text{and} & Z & = & (X \cap Y) \cup (X - B) \cup (Y - A) & \\
\text{assuming} & X \subseteq A & & \text{and} \quad Y \subseteq B & (1.22) \\
\Box_{b:B}\,(b \to P_b) \;|||\; \Box_{c:C}\,(c \to Q_c) & & = & (\Box_{b:B}\,(b \to (P_b \;|||\; \Box_{c:C}\,(c \to Q_c)))) \;\Box & \\
 & & & (\Box_{c:C}\,(c \to (Q_c \;|||\; \Box_{b:B}\,(b \to P_c)))) & (1.23)
\end{array}
$$

CSP

Axiomatic laws (cont'd)

$$
\begin{array}{rcll}
SKIP \setminus x & = & SKIP & (1.24) \\
STOP \setminus x & = & STOP & (1.25) \\
(P \setminus x) \setminus y & = & (P \setminus y) \setminus x & (1.26) \\
(x \to P) \setminus x & = & P \setminus x & (1.27) \\
(x \to P) \setminus y & = & x \to (P \setminus y) \quad \text{if} \quad x \neq y & (1.28) \\
(P \,;\, Q) \setminus x & = & (P \setminus x) \,;\, (Q \setminus x) & (1.29) \\
(P \;[\![A \,|\, B]\!]\; Q) \setminus x & = & P \;[\![A \,|\, B - \{x\}]\!]\; (Q \setminus x) & \\
 & & \text{if} \quad x \notin A & (1.30) \\
(P \sqcap Q) \setminus x & = & (P \setminus x) \sqcap (Q \setminus x) & (1.31) \\
((x \to P) \,\Box\, (y \to Q)) \setminus x & = & (P \setminus x) \sqcap ((P \setminus x) \,\Box\, (y \to (Q \setminus x))) & \\
 & & \text{if} \quad x \neq y & (1.32)
\end{array}
$$

CSP

Axiomatic laws (cont'd)

$$f(STOP) = STOP \tag{1.33}$$
$$f(e \rightarrow P) = f(e) \rightarrow f(P) \tag{1.34}$$
$$f(P\,;\,Q) = f(P)\,;f(Q) \quad \text{if} \quad f^{-1}(\surd) = \{\surd\} \tag{1.35}$$
$$f(P \;|||\; Q) = f(P) \;|||\; f(Q) \tag{1.36}$$
$$f(P \,\Box\, Q) = f(P) \,\Box\, f(Q) \tag{1.37}$$
$$f(P \sqcap Q) = f(P) \sqcap f(Q) \tag{1.38}$$
$$f(P \setminus f^{-1}(x)) = f(P) \setminus x \tag{1.39}$$

CSP

Basic definitions (examples)

$$\{\langle coffee,coffee,coffee\rangle, \langle coin,tea\rangle\} \subset traces(TD)$$

- Catenation: $s^\frown t$

$$\langle s_1, s_2, .., s_m\rangle^\frown\langle t_1, t_2, .., t_n\rangle = \langle s_1, .., s_m, t_1, .., t_n\rangle$$

- Restriction: $s \upharpoonright B$, trace s restricted to elements of set B

Example: $\langle a, b, c, d, b, d, a\rangle \upharpoonright \{a, b, c\} = \langle a, b, c, b, a\rangle$

- Replication: s^n trace s repeated n times.

Example: $\langle a, b\rangle^2 = \langle a, b, a, b\rangle$

- Count: $s \downarrow x$ number of occurrences of event x in trace s

Example: $\langle x, y, z, x, x\rangle \downarrow x = 3$

- Length: $|s|$ the length of trace s.

Example: $|\langle a, b, c\rangle| = 3$

- Merging: $merge(s, t)$ the set of all possible interleavings of trace s with trace t

$$(\langle coin,tea,coin,tea,coin,coin\rangle, \{tea,coin\}) \in failures(VM)$$

$$\langle\rangle \in divergences(CLOCK \setminus tick)$$

CSP

Example of requested properties

$$\forall (s, X) : failures(P). \quad s \downarrow in > s \downarrow out \Longrightarrow out \notin X$$

CSP

Properties of the set of failures

(1) $(\langle\rangle, \{\}) \in F$

(2) $(s^\frown t, \{\}) \in F \Longrightarrow (s, \{\}) \in F$

(3) $(s, Y) \in F \wedge X \subseteq Y \Longrightarrow (s, X) \in F$

(4) $(s, X) \in F \wedge (\forall c \in Y.((s^\frown\langle c\rangle, \{\}) \notin F)) \Longrightarrow (s, X \cup Y) \in F$

(5) $(\forall Y \in p(X).(s, Y) \in F) \Longrightarrow (s, X) \in F$

(6) $s \in D \wedge t \in \Sigma^* \Longrightarrow s^\frown t \in D$

(7) $s \in D \wedge X \subseteq \Sigma \Longrightarrow (s, X) \in F$

CSP

Failure-Divergence semantics

$$(F_1, D_1) \sqsubseteq (F_2, D_2) \iff F_1 \supseteq F_2 \wedge D_1 \supseteq D_2$$

The system P_1 is worse than P_2 : it can deadlock or diverge whenever P_2 can.

One particular process : the chaos process (= bottom of the complete lattice)

$$\begin{aligned} failures(\bot) &= \Sigma^* \times \mathbf{P}\,\Sigma \\ divergences(\bot) &= \Sigma^* \end{aligned}$$

Least fixpoint computation

$$\mu\, X \bullet F(X) = \sqcup\{F^n(\bot) | n \in \mathbf{N}\}$$

CSP

Failure-Divergence semantics

$$\begin{aligned}
divergences(STOP) &= \{\} \\
failures(STOP) &= \{\langle\rangle\} \times \mathbf{P}\,\Sigma \\
divergences(SKIP) &= \{\} \\
failures(SKIP) &= (\{\langle\rangle\} \times \mathbf{P}(\Sigma - \surd)) \\
&\cup\ (\{\langle\surd\rangle\} \times \mathbf{P}\,\Sigma) \\
divergences(x \rightarrow P) &= \{\langle x\rangle^\frown s | s \in divergences(P)\} \\
failures(x \rightarrow P) &= \{(\langle\rangle, X) | X \subseteq \Sigma - \{x\}\} \\
&\cup\ \{(\langle x\rangle^\frown s, X) | (s, X) \in failures(P)\} \\
divergences(P\,;\,Q) &= divergences(P) \\
&\cup\ \left\{ \begin{array}{c} s^\frown t | s^\frown\langle\surd\rangle \in traces(P) \wedge s\surd\text{-}free \\ \wedge t \in divergences(Q) \end{array} \right\} \\
failures(P\,;\,Q) &= \{(s, X) | s\surd\text{-}free \wedge (s, X \cup \langle\surd\rangle) \in failures(P)\} \\
&\cup\ \left\{ \begin{array}{c} (s^\frown t, X) | s^\frown\langle\surd\rangle \in traces(P) \wedge s\surd\text{-}free \wedge \\ (t, X) \in failures(Q) \end{array} \right\} \\
&\cup\ \{(s, X) | s \in divergences(P\,;\,Q)\}
\end{aligned}$$

CSP

Failure-Divergence semantics

$$
\begin{array}{rcl}
\begin{array}{c} divergences \\ (P \ [\![A \,|\, B]\!] \ Q) \end{array} & = & \left\{ s^\frown t | s \in (A \cup B \cup \{\surd\})^* \wedge \left(\begin{array}{c} \left(\begin{array}{c} s \upharpoonright (A \cup \{\surd\}) \in divergences(P) \wedge \\ s \upharpoonright (B \cup \{\surd\}) \in traces(Q) \end{array} \right) \\ \vee \left(\begin{array}{c} s \upharpoonright (B \cup \{\surd\}) \in divergences(Q) \wedge \\ s \upharpoonright (A \cup \{\surd\}) \in traces(P) \end{array} \right) \end{array} \right) \right\} \\
failures(P \ [\![A \,|\, B]\!] \ Q) & = & \left\{ \begin{array}{c} (s, X \cup Y \cup Z) | s \in (A \cup B \cup \{\surd\})^* \\ \wedge X \subseteq (A \cup \{\surd\}) \wedge Y \subseteq (B \cup \{\surd\}) \wedge \\ Z \subseteq (\Sigma - (A \cup B \cup \{\surd\})) \\ \wedge (s \upharpoonright (A \cup \{\surd\}), X) \in failures(P) \\ \wedge (s \upharpoonright (B \cup \{\surd\}), Y) \in failures(Q) \end{array} \right\} \\
 & \cup & \{(s, X) | s \in divergences(P \ [\![A \,|\, B]\!] \ Q)\}
\end{array}
$$

CSP

Failure-Divergence semantics

$$
\begin{array}{rcl}
divergences(P \ ||| \ Q) & = & \left\{ \begin{array}{c} \exists s, t. \quad u \in merge(s, t) \wedge \\ \left(\begin{array}{c} (s \in divergences(P) \wedge t \in traces(Q)) \vee \\ (s \in traces(P) \wedge t \in divergences(Q)) \end{array} \right) \end{array} \right\} \\
failures(P \ ||| \ Q) & = & \left\{ \begin{array}{c} (u, X) | \exists s, t. \\ \left(\begin{array}{c} \left(\begin{array}{c} (s, X - \{\surd\}) \in failures(P) \wedge \\ (t, X) \in failures(Q) \end{array} \right) \vee \\ \left(\begin{array}{c} (s, X) \in failures(P) \wedge \\ (t, X - \{\surd\}) \in failures(Q) \end{array} \right) \end{array} \right) \wedge \\ u \in merge(s, t) \end{array} \right\} \\
 & \cup & \{(s, X) | s \in divergences(P \ ||| \ Q)\} \\
divergences(P \sqcap Q) & = & divergences(P) \cup divergences(Q) \\
failures(P \sqcap Q) & = & failures(P) \cup failures(Q)
\end{array}
$$

CSP

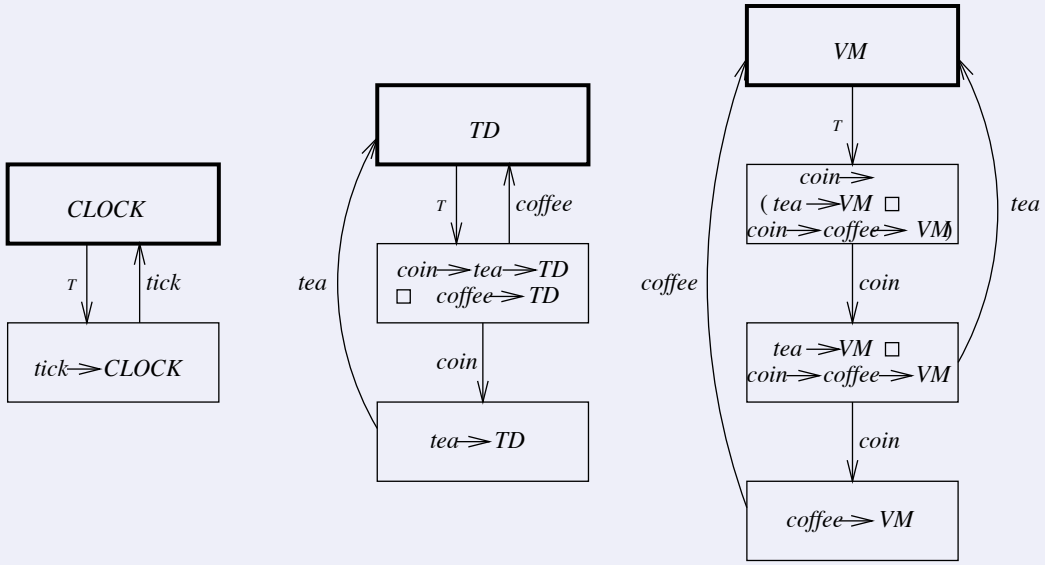
Failure-Divergence semantics

$$
\begin{aligned}
divergences(P \,\Box\, Q) &= divergences(P) \cup divergences(Q) \\
failures(P \,\Box\, Q) &= \left\{ \begin{array}{c} (s,X) | (s,X) \in failures(P) \cap failures(Q) \vee \\ \left(\begin{array}{c} s \neq \langle\rangle \wedge \\ (s,X) \in failures(P) \cup failures(Q) \end{array} \right) \end{array} \right\} \\
&\cup \{(s,X) | s \in divergences(P \,\Box\, Q)\} \\
divergences(P \setminus x) &= \left\{ \begin{array}{c} (s \upharpoonright (\Sigma - \{x\}))^\frown t | \\ \left(\begin{array}{c} s \in divergences(P) \\ \vee (\forall n . s^\frown \langle x \rangle^n \in traces(P)) \end{array} \right) \end{array} \right\} \\
failures(P \setminus x) &= \{(s \upharpoonright (\Sigma - \{x\}), X) | (s, X \cup \{x\}) \in failures(P)\} \\
&\cup \{(s,X) | s \in divergences(P \setminus \{x\})\} \\
divergences(f(P)) &= \{f(s)t | s \in divergences(P)\} \\
failures(f(P)) &= \{(f(s), X) | (s, f^{-1}(X)) \in failures(P)\} \\
&\cup \{(s,X) | s \in divergences(f(P))\}
\end{aligned}
$$

CSP

Operational semantics : example

LTS where the current process identifies the global state



CSP

Operational semantics

Primitive processes:

$$\frac{}{SKIP \xrightarrow{\surd} STOP}$$

Prefix:

$$\frac{}{(a \rightarrow P) \xrightarrow{a} P}$$

External choice:

$$\frac{P \xrightarrow{a} P'}{(P \,\Box\, Q) \xrightarrow{a} P'} a \neq \tau$$

$$\frac{Q \xrightarrow{a} Q'}{(P \,\Box\, Q) \xrightarrow{a} Q'} a \neq \tau$$

$$\frac{P \xrightarrow{\tau} P'}{(P \,\Box\, Q) \xrightarrow{\tau} (P' \,\Box\, Q)}$$

$$\frac{Q \xrightarrow{\tau} Q'}{(P \,\Box\, Q) \xrightarrow{\tau} (P \,\Box\, Q')}$$

CSP

Operational semantics

Internal choice:

$$\frac{}{(P \sqcap Q) \xrightarrow{\tau} P}$$

$$\frac{}{(P \sqcap Q) \xrightarrow{\tau} Q}$$

Sequential Composition:

$$\frac{P \xrightarrow{a} P'}{(P \,;\, Q) \xrightarrow{a} (P' \,;\, Q)} a \neq \surd$$

$$\frac{P \xrightarrow{\surd} P'}{(P \,;\, Q) \xrightarrow{\tau} Q}$$

CSP

Operational semantics

Parallel Composition:

$$\frac{P \xrightarrow{a} P'}{P \ [\![A \,|\, B]\!] \ Q \xrightarrow{a} P' \ [\![A \,|\, B]\!] \ Q} a \in (A - B - \{\surd\}) \cup \{\tau\}$$

$$\frac{Q \xrightarrow{a} Q'}{P \ [\![A \,|\, B]\!] \ Q \xrightarrow{a} P \ [\![A \,|\, B]\!] \ Q'} a \in (B - A - \{\surd\}) \cup \{\tau\}$$

$$\frac{P \xrightarrow{a} P' \quad Q \xrightarrow{a} Q'}{P \ [\![A \,|\, B]\!] \ Q \xrightarrow{a} P' \ [\![A \,|\, B]\!] \ Q'} a \in (A \cap B) \cup \{\surd\}$$

Interleaving:

$$\frac{P \xrightarrow{a} P'}{P \ ||| \ Q \xrightarrow{a} P' \ ||| \ Q} a \neq \surd$$

$$\frac{Q \xrightarrow{a} Q'}{P \ ||| \ Q \xrightarrow{a} P \ ||| \ Q'} a \neq \surd$$

$$\frac{P \xrightarrow{\surd} P' \quad Q \xrightarrow{\surd} Q'}{P \ ||| \ Q \xrightarrow{\surd} P' \ ||| \ Q'}$$

CSP

Operational semantics

Hiding:

$$\frac{P \xrightarrow{a} P'}{(P \setminus A) \xrightarrow{\tau} (P' \setminus A)} a \in A \cup \{\tau\}$$

$$\frac{P \xrightarrow{a} P'}{(P \setminus A) \xrightarrow{a} (P' \setminus A)} a \notin A \cup \{\tau\}$$

Alphabet Transformation:

$$\frac{P \xrightarrow{a} P'}{f(P) \xrightarrow{f(a)} f(P')}$$

Recursion:

$$\frac{}{\mu X \bullet F(X) \xrightarrow{\tau} F(\mu X \bullet F(X))}$$

CSP

Extentions

- Parameterized processes

$$BUFF(in, out) = in \rightarrow out \rightarrow BUFF(in, out)$$

- type of a channel

$$type(c) = \{v | c.v \in \Sigma\}$$

CSP

Translation extended CSP into CSP

$$(c!v \rightarrow P) = (c.v \rightarrow P)$$

$$(c?x \rightarrow P(x)) = \Box_{v:type(c)} (c.v \rightarrow P(v))$$

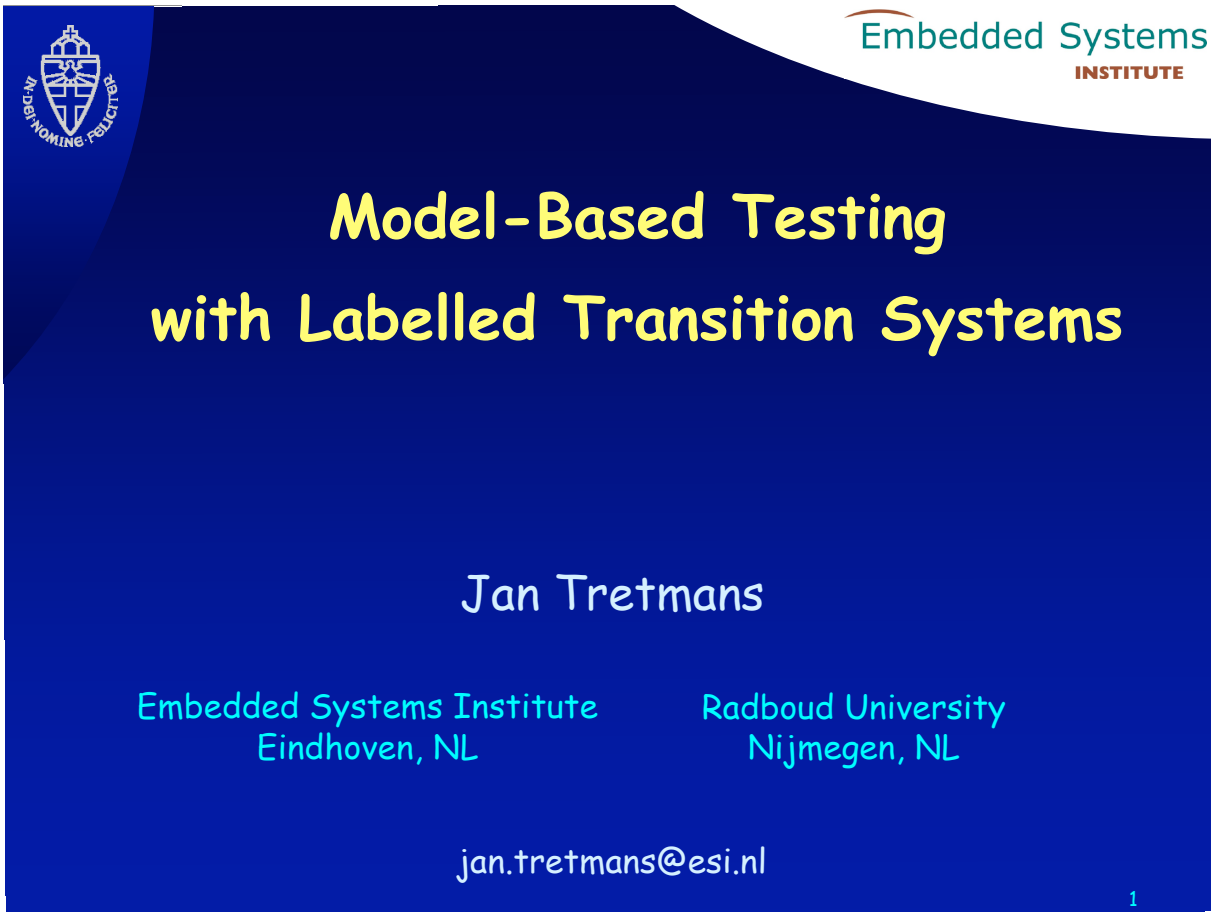
Chapter 9 : Testing

Plan

Conformance Tests synthesis for reactive systems

Note

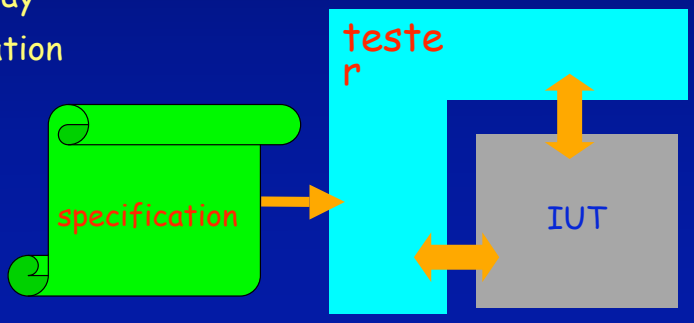
These slides have been given by Jan Tretmans in 2006 during a seminar in Rennes



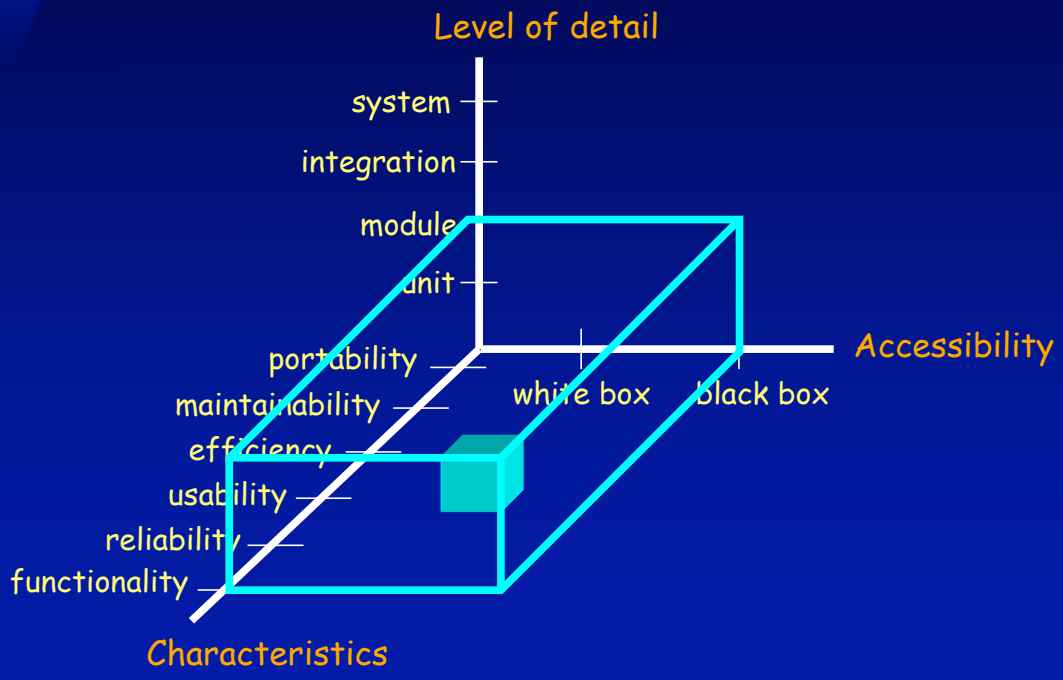
Testing

Testing:

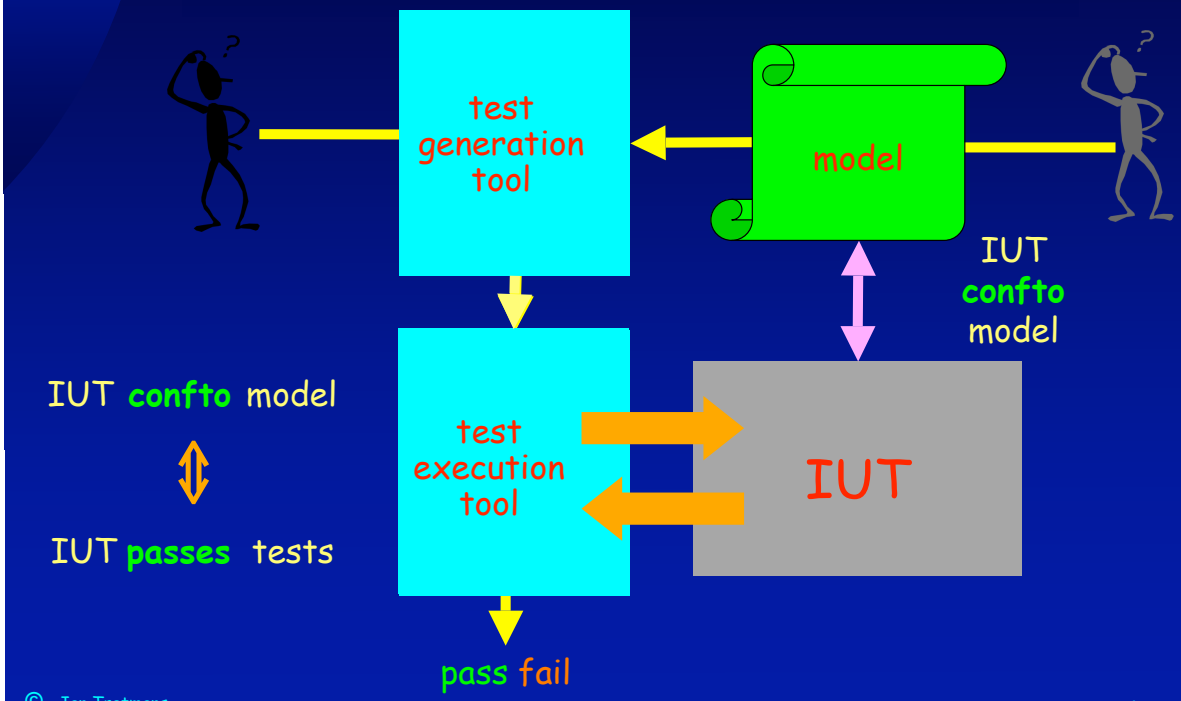
checking or measuring some quality characteristics
of an executing object
by performing experiments
in a controlled way
w.r.t. a specification



Types of Testing



Automated Model-Based Testing

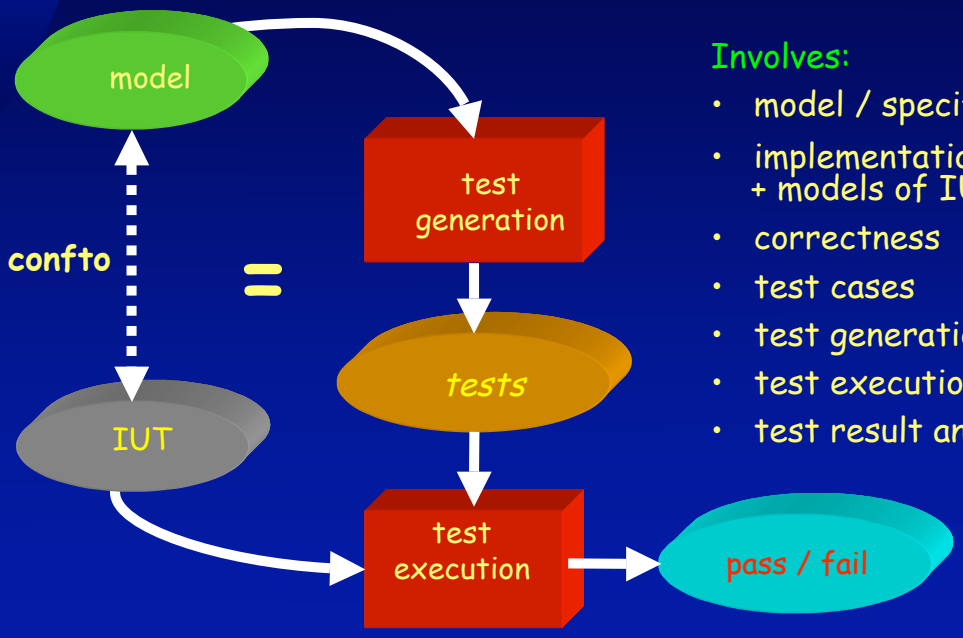


Approaches to Model-Based Testing

Several modelling paradigms:

- Finite State Machine
- Pre/post-conditions
- Labelled Transition Systems
- Programs as Functions
- Abstract Data Type testing
- .....

Model-Based Testing for LTS



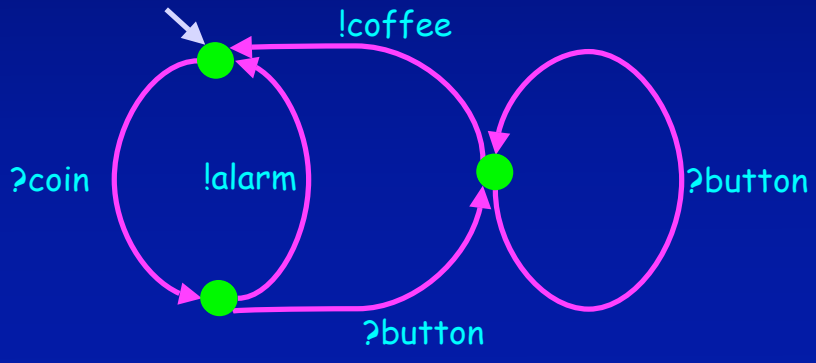
Involves:

- model / specification
- implementation IUT + models of IUTs
- correctness
- test cases
- test generation
- test execution
- test result analysis

Models of Specifications: Labelled Transition Systems

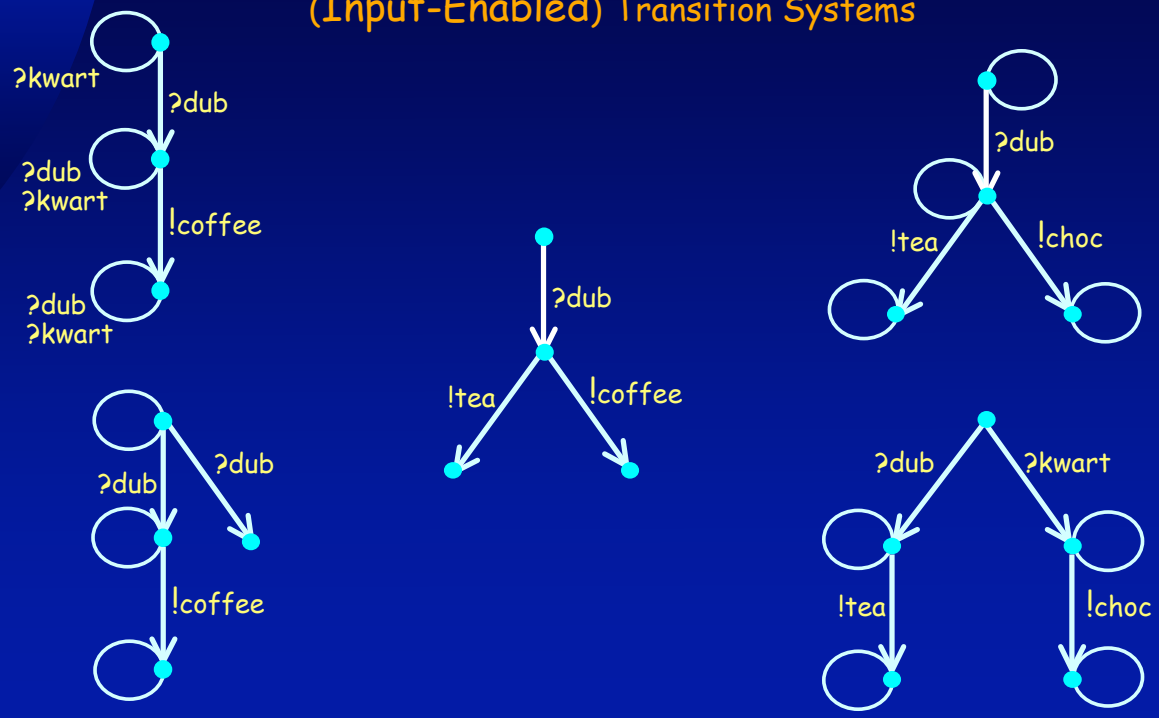
Labelled Transition System $\langle S, L, T, s_0 \rangle$

- S: states
- L: actions
- T: transitions $T \subseteq S \times (L \cup \{\tau\}) \times S$
- s_0: initial state $s_0 \in S$



Example Models

(Input-Enabled) Transition Systems



Correctness

Implementation Relation **ioco**

i **ioco** s $=_{def}$ $\forall \sigma \in Straces(s)$: $out(i \textbf{ after } \sigma) \subseteq out(s \textbf{ after } \sigma)$

Intuition:

i **ioco**-conforms to s, iff

- if i produces output x after trace σ,
  then s can produce x after σ
- if i cannot produce any output after trace σ,
  then s cannot produce any output after σ (*quiescence* δ)

Correctness

Implementation Relation **ioco**

i **ioco** s $=_{def}$ $\forall \sigma \in Straces(s) : out(i \textbf{ after } \sigma) \subseteq out(s \textbf{ after } \sigma)$

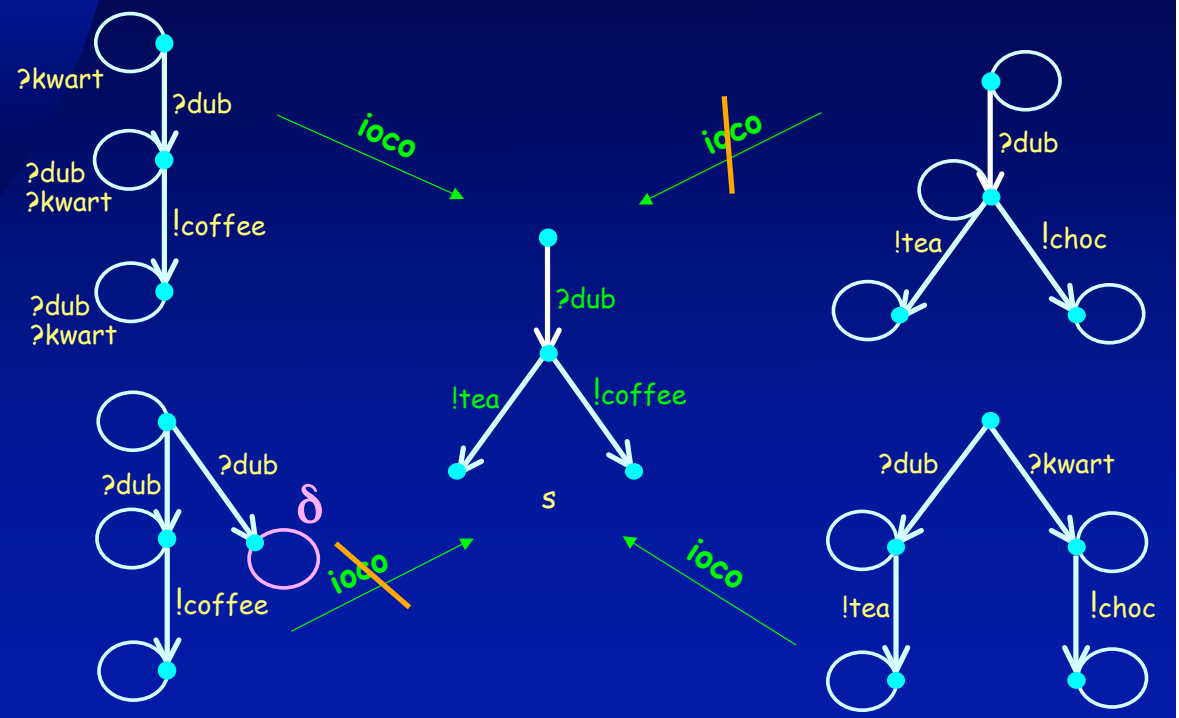
$p \xrightarrow{\delta} p$ = $\forall\ !x \in L_U \cup \{\tau\} .\ p \not\xrightarrow{!x}$

$Straces(s)$ = $\{\ \sigma \in (L \cup \{\delta\})^* \mid s \stackrel{\sigma}{\Longrightarrow}\ \}$

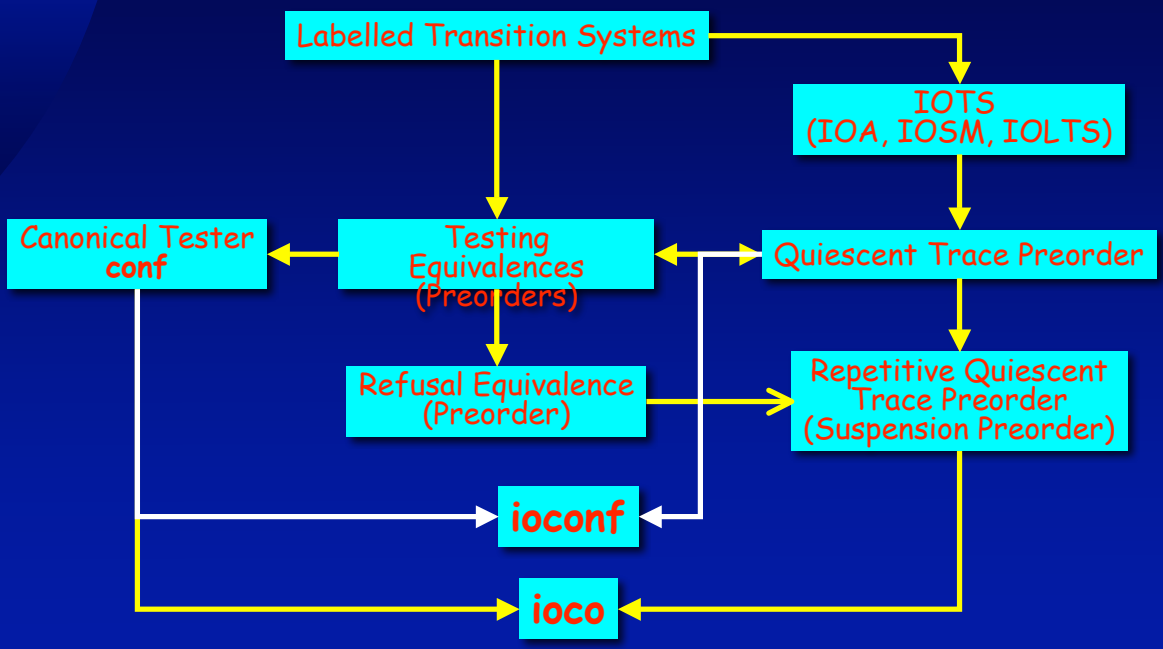
p **after** σ = $\{\ p' \mid p \stackrel{\sigma}{\Longrightarrow} p'\ \}$

$out(P)$ = $\{\ !x \in L_U \mid p \xrightarrow{!x},\ p \in P\ \} \cup \{\ \delta \mid p \xrightarrow{\delta} p,\ p \in P\ \}$

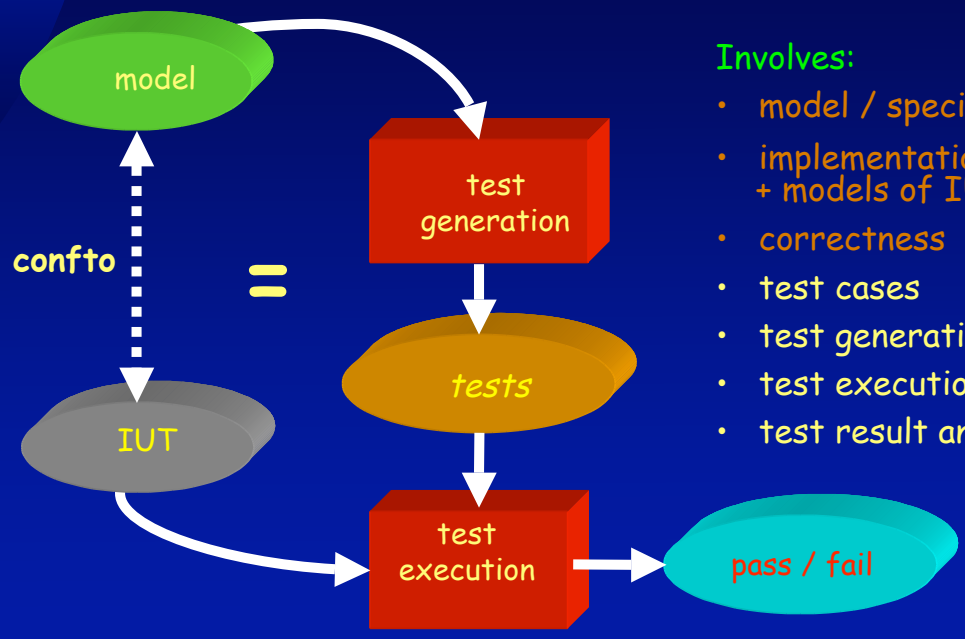
Implementation Relation **ioco**



Genealogy of ioco



Model-Based Testing for LTS



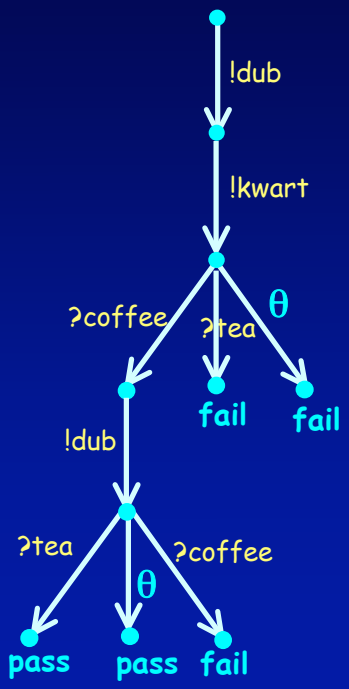
Involves:

- model / specification
- implementation IUT + models of IUTs
- correctness
- test cases
- test generation
- test execution
- test result analysis

Test Cases

Model of a test case
= transition system :

- 'quiescence' label θ
- tree-structured
- finite, deterministic
- final states **pass** and **fail**
- from each state ≠ **pass**, **fail** :
  - either one input !a
  - or all outputs ?x and θ



ioco Test Generation Algorithm

Algorithm

To generate a test case from transition system specification s_0 compute T(S), with S a set of states, and initially S = s_0 after ε ;

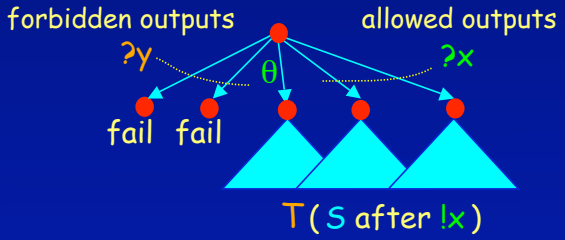
For T(S), apply the following recursively, non-deterministically:

1 end test case

pass

2 supply input

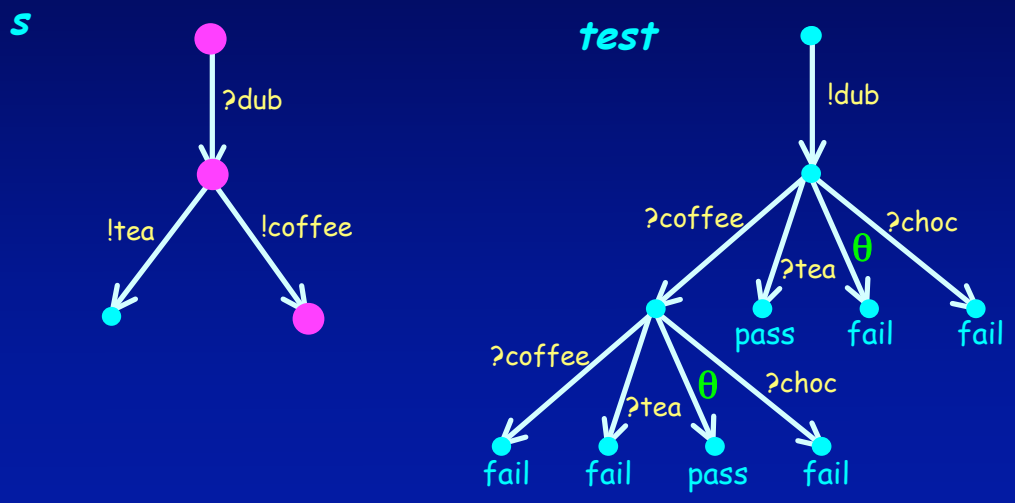
3 observe output



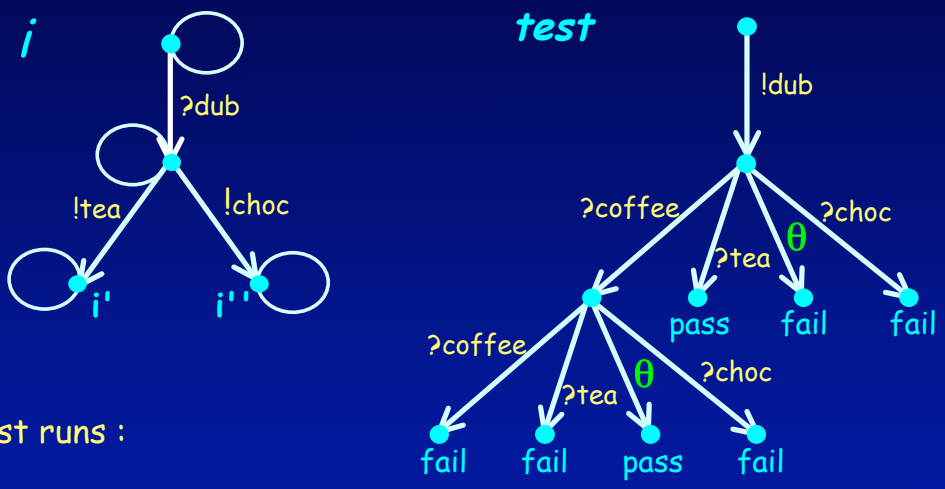
allowed outputs or δ: $!x \in out(S)$

forbidden outputs or δ: $!y \notin out(S)$

Test Generation Example



Test Execution Example



Two test runs :

t ‖ i $\xrightarrow{\text{dub tea}}$ pass ‖ i'

t ‖ i $\xrightarrow{\text{dub choc}}$ fail ‖ i''

i fails t

Test Result Analysis

Completeness of **ioco** Test Generation

For every test t generated with algorithm we have:

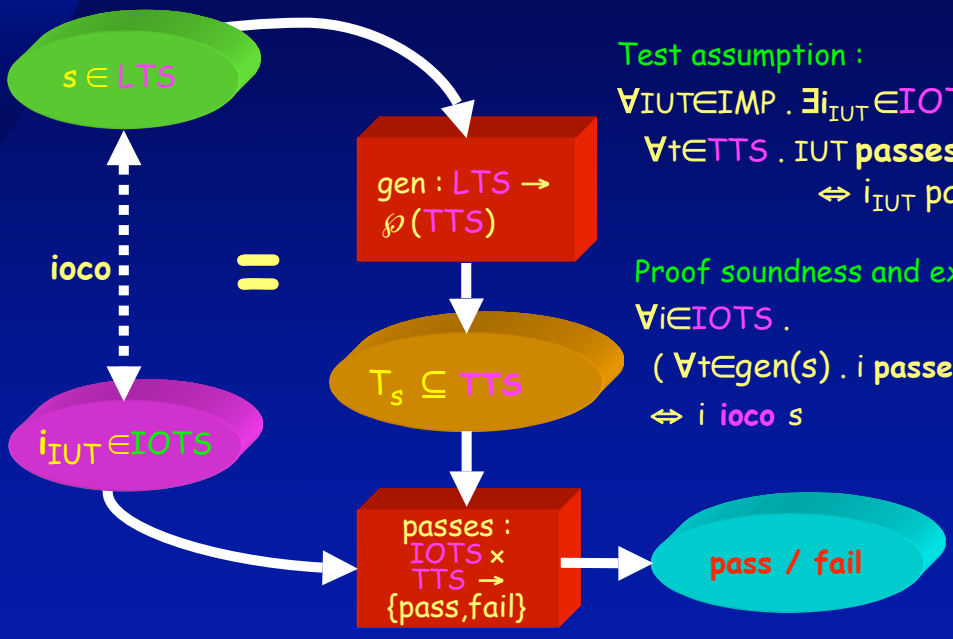
- Soundness :
  t will never fail with correct implementation

  i **ioco** s implies i passes t

- Exhaustiveness :
  each incorrect implementation can be detected with a generated test t

  i ~~**ioco**~~ s implies $\exists$ t : i fails t

Formal Testing with Transition Systems



Test assumption :
$\forall IUT \in IMP . \exists i_{IUT} \in IOTS .$
$\forall t \in TTS .$ IUT **passes** t
$\Leftrightarrow i_{IUT}$ passes t

Proof soundness and exhaustiveness:
$\forall i \in IOTS .$
($\forall t \in gen(s) .$ i **passes** t)
$\Leftrightarrow$ i **ioco** s

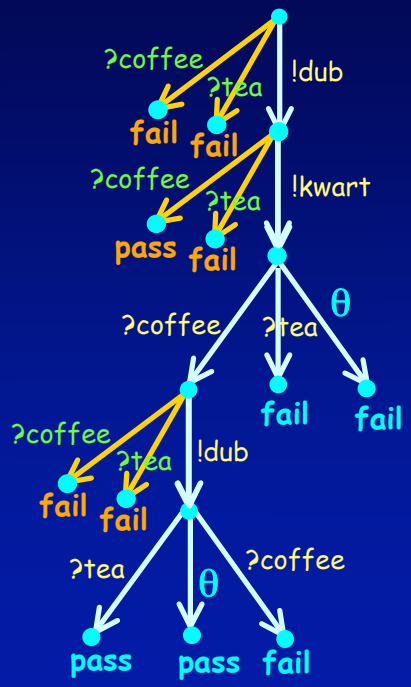
Issues with ioco LTS Testing

- Compositional/component-based testing
- Under-specification
- State-space explosion: symbolic representations for data, ...
- (Non-) Input enabledness
- Test assumption (hypothesis)
- Real-time, hybrid extensions
- Action refinement
- Paradox of test-input enabledness
- . . . . .

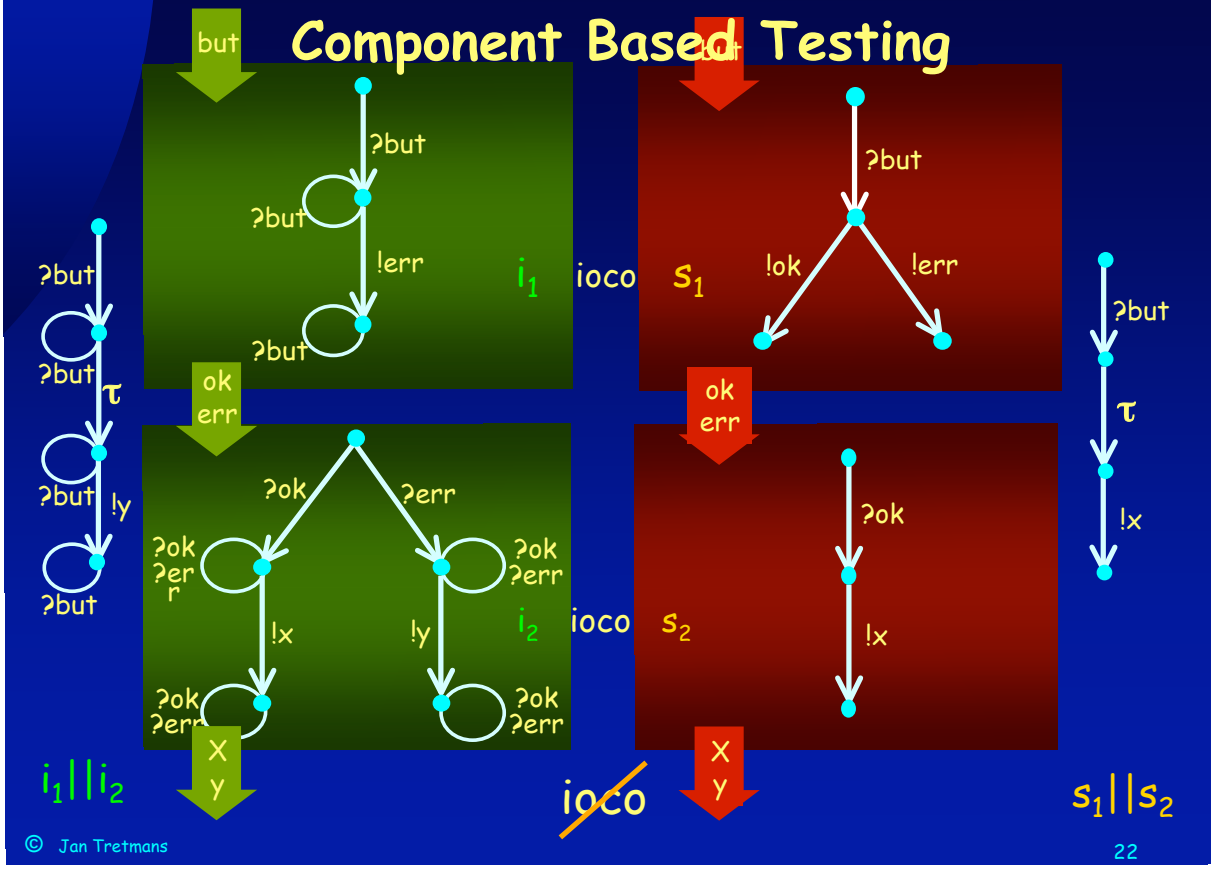
"Output" Enabled Test Cases

Model of a test case
= transition system :

- 'quiescence' label θ
- tree-structured
- finite, deterministic
- final states **pass** and **fail**
- from each state ≠ **pass**, **fail** :
  - either one input !a
  - or all outputs ?x and θ

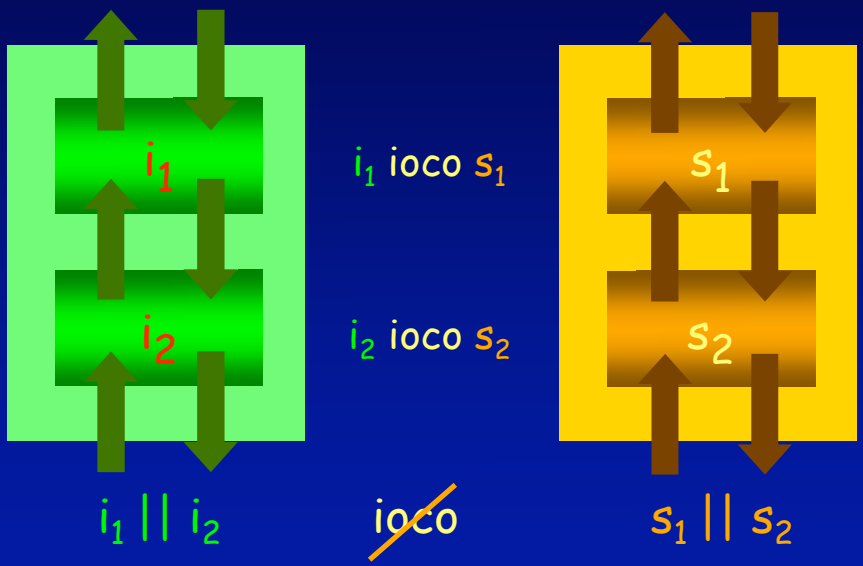


Component Based Testing



Compositional Testing

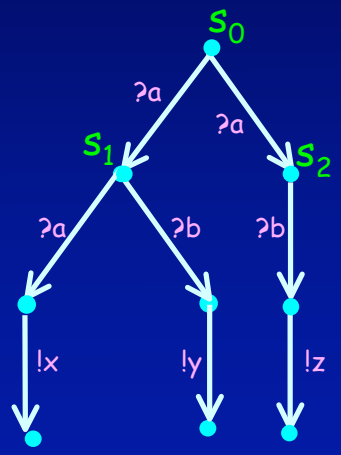
Component Based Testing



If s_1, s_2 input enabled - s_1, $s_2 \in$ IOTS - then ioco is preserved !

Underspecification: uioco

i **ioco** $s \;\Leftrightarrow\; \forall \sigma \in Straces(s) : out(i \text{ after } \sigma) \subseteq out(s_0 \text{ after } \sigma)$



$out(s_0 \text{ after } ?b) = \emptyset$

but $?b \notin Straces(s)$: under-specification : anything allowed after ?b

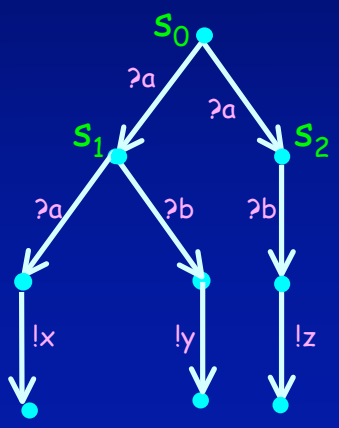
$out(s_0 \text{ after } ?a\ ?a) = \{ !x \}$

and $?a\ ?a \in Straces(s)$

but from s_2, $?a\ ?a$ is under-specified : anything allowed after ?a ?a ?

Underspecification: uioco

i **uioco** $s \;\Leftrightarrow\; \forall \sigma \in Utraces(s) : out(i \text{ after } \sigma) \subseteq out(s_0 \text{ after } \sigma)$



$Utraces(s) =$

$\{\ \sigma \in Straces(s)\ \mid\ \forall\, \sigma_1\ ?a\ \sigma_2 = \sigma,$

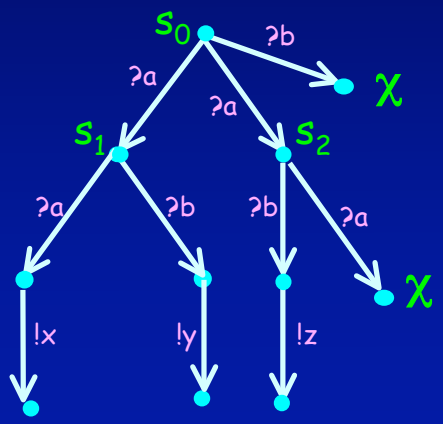
$\forall\, s' :\ s \stackrel{\sigma_1}{\Longrightarrow} s' \Rightarrow s' \stackrel{?a}{\Longrightarrow}\ \}$

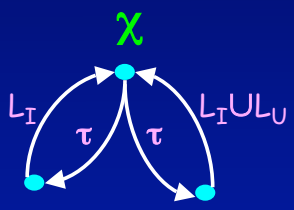
Now s is under-specified in s_2 for ?a : anything is allowed.

ioco $\subset$ **uioco**

Underspecification: uioco

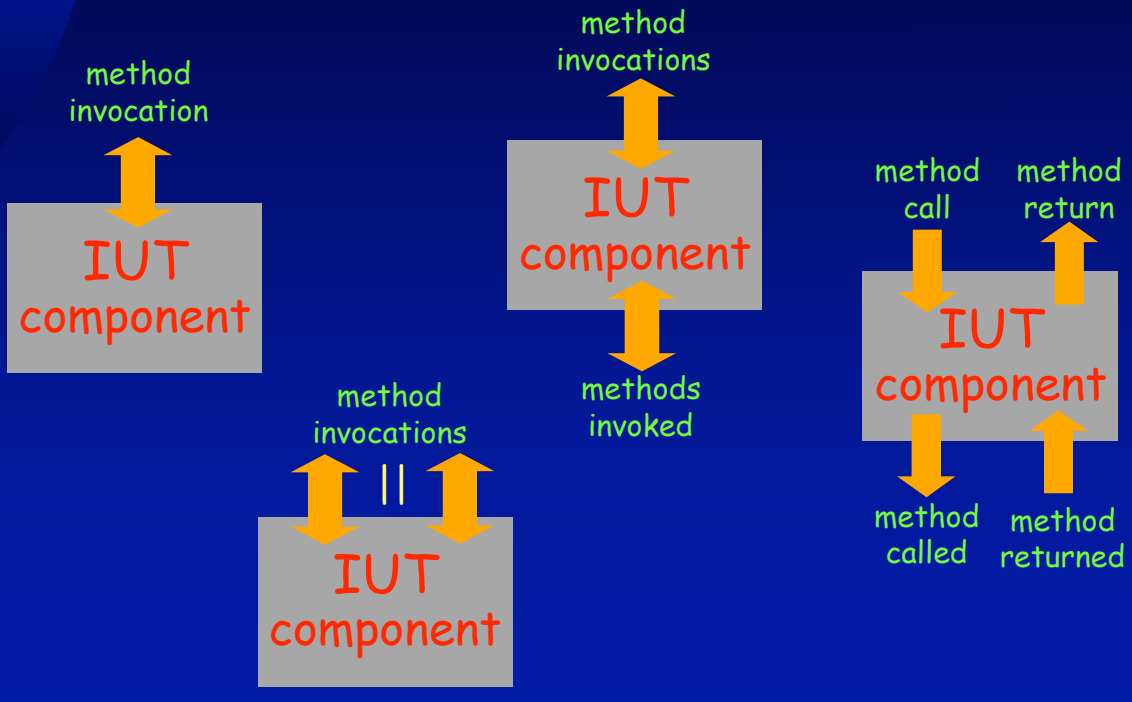
i **uioco** $s \Leftrightarrow \forall \sigma \in Utraces(s) : out(i \text{ after } \sigma) \subseteq out(s_0 \text{ after } \sigma)$



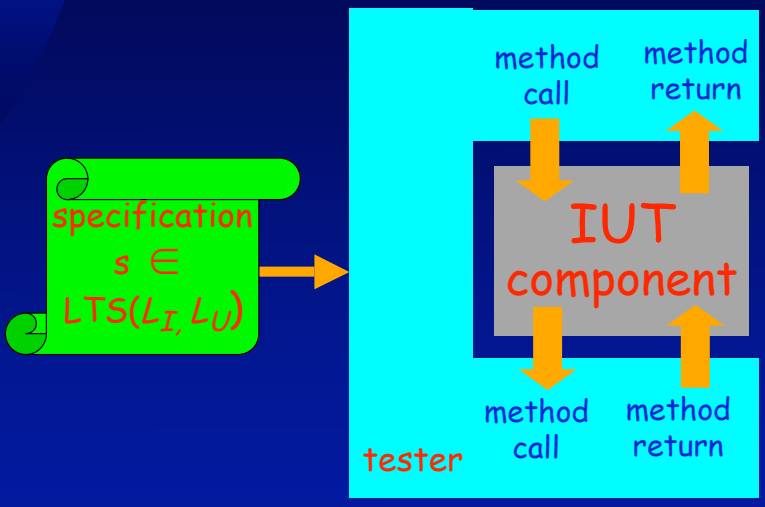


Alternatively, via chaos process χ for under-specified inputs

Testing Components



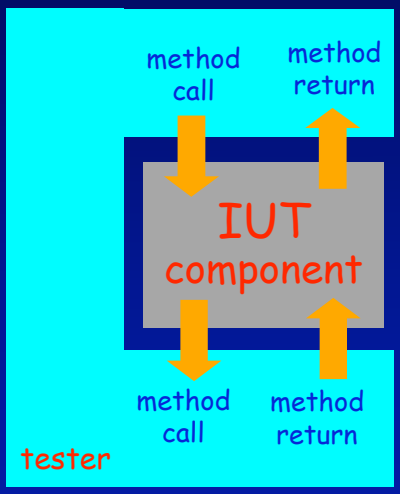
Testing Components



L_I = offered methods calls ∪ used methods returns

L_U = offered methods returns ∪ used methods calls

Testing Components



Input-enabledness:

∀ s of IUT, ∀ ?a ∈ L_I : s $\xRightarrow{?a}$

No ! ?

Correctness

Implementation Relation **wioco**

i **uioco** s $=_{def}$ $\forall \sigma \in Utraces(s) : out(i \text{ after } \sigma) \subseteq out(s \text{ after } \sigma)$

i **wioco** s $=_{def}$ $\forall \sigma \in Utraces(s) : out(i \text{ after } \sigma) \subseteq out(s \text{ after } \sigma)$
and $in(i \text{ after } \sigma) \supseteq in(s \text{ after } \sigma)$

$in(s \text{ after } \sigma) = \{ a? \in L_I \mid s \text{ after } \sigma \text{ must } a? \}$

$s \text{ after } \sigma \text{ must } a? = \forall s' \; (s \stackrel{\sigma}{\Longrightarrow} s' \Rightarrow s' \stackrel{a?}{\Longrightarrow})$

Variations on a Theme

i **ioco** $s \Leftrightarrow \forall \sigma \in Straces(s) : out(i \text{ after } \sigma) \subseteq out(s \text{ after } \sigma)$

$i \leq_{ior} s \Leftrightarrow \forall \sigma \in (L \cup \{\delta\})^* : out(i \text{ after } \sigma) \subseteq out(s \text{ after } \sigma)$

i **ioconf** $s \Leftrightarrow \forall \sigma \in traces(s) : out(i \text{ after } \sigma) \subseteq out(s \text{ after } \sigma)$

i **ioco**$_F$ $s \Leftrightarrow \forall \sigma \in F : out(i \text{ after } \sigma) \subseteq out(s \text{ after } \sigma)$

i **uioco** $s \Leftrightarrow \forall \sigma \in Utraces(s) : out(i \text{ after } \sigma) \subseteq out(s \text{ after } \sigma)$

i **mioco** s multi-channel ioco

i **wioco** s non-input-enabled ioco

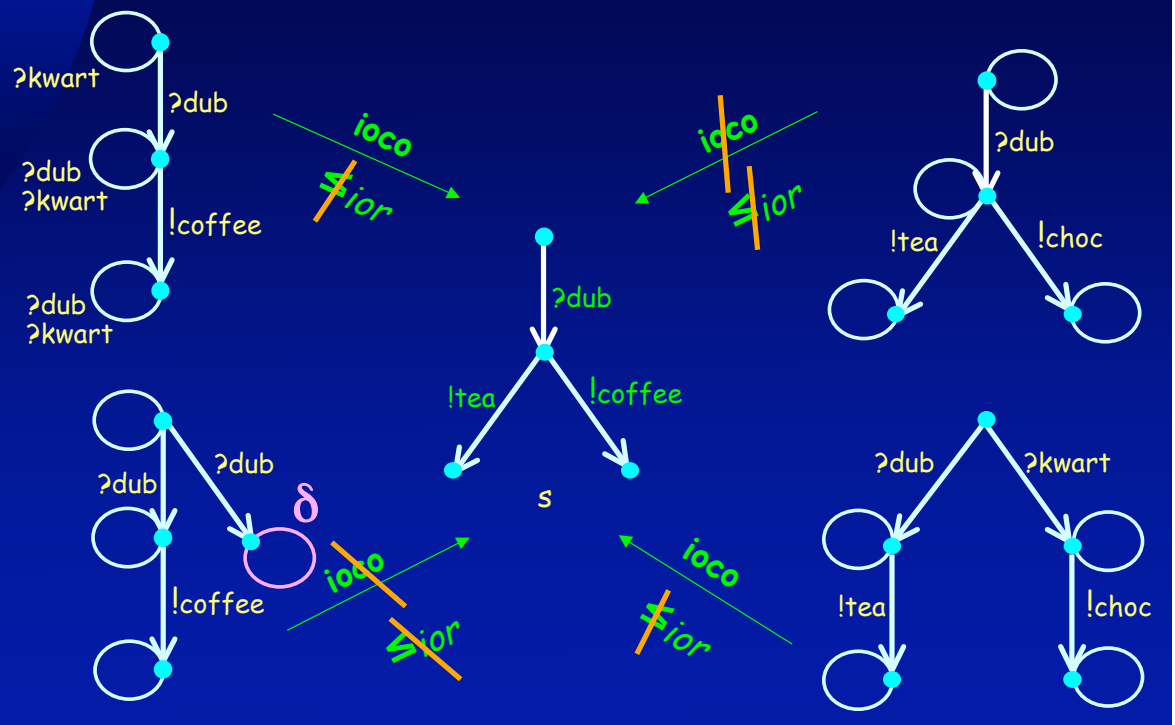
i **sioco** s symbolic ioco

i **(r)tioco** s (real) timed tioco (Aalborg, Twente, Grenoble, Bordeaux,. . . .)

i **ioco**$_r$ s refinement ioco

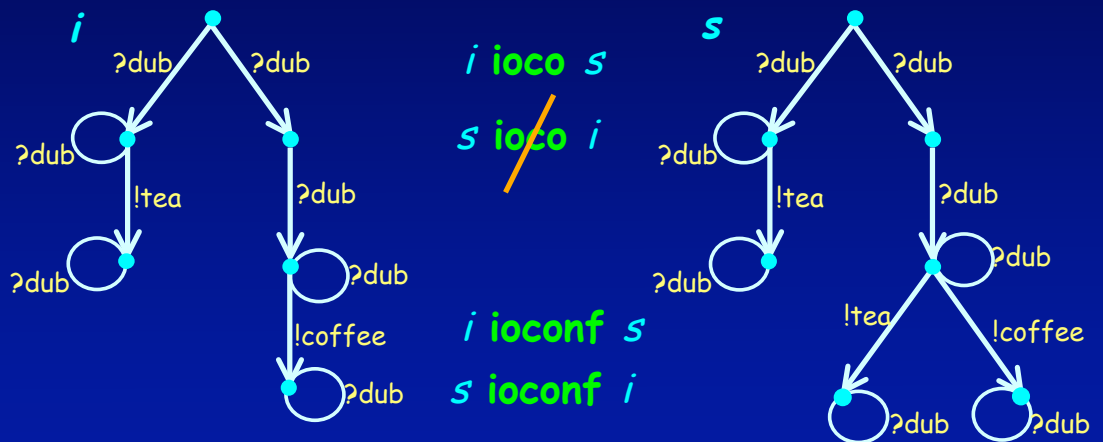
i **hioco** s hybrid ioco

Implementation Relation ioco



Implementation Relation ioco

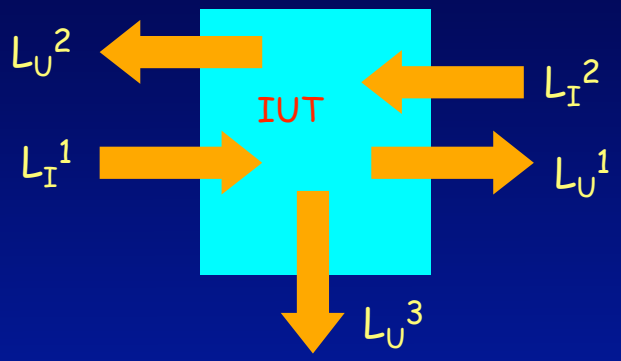
i **ioco** s $=_{def}$ $\forall \sigma \in Straces(s)$: $out(i \textbf{ after } \sigma) \subseteq out(s \textbf{ after } \sigma)$



out (*i* **after** ?dub.?dub) = *out* (*s* **after** ?dub.?dub) = { !tea, !coffee }

out (*i* **after** ?dub.δ.?dub) = { !coffee } ≠ *out* (*s* **after** ?dub.δ.?dub) = { !tea, !coffee }

Variations on a Theme: mioco



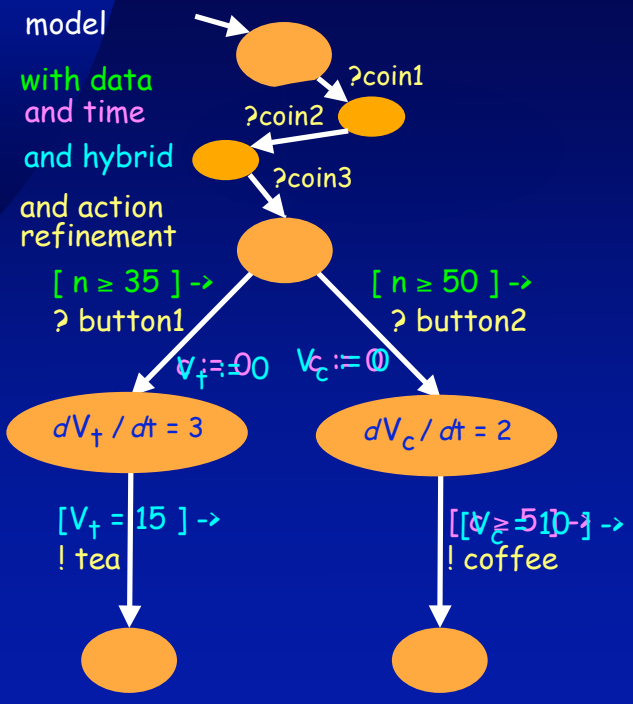
i **mioco** s $\Leftrightarrow \forall \sigma \in Straces'(s) : out'(i \text{ after } \sigma) \subseteq out'(s \text{ after } \sigma)$

$p \xrightarrow{\delta_k} p = \forall\, !x \in L^k \cup \{\tau\} .\ p \not\xrightarrow{!x}$

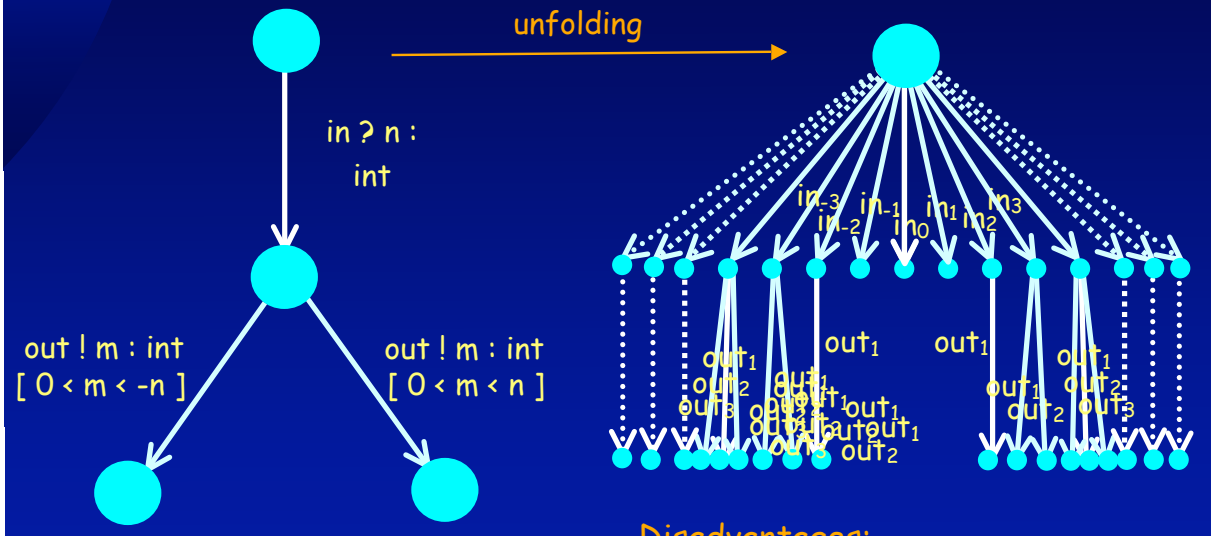
$Straces'(s) = \{\ \sigma \in (L \cup \{\delta_k\})^* \mid s \stackrel{\sigma}{\Longrightarrow}\ \}$

$out'(P) = \{\ !x \in L_U \mid p \xrightarrow{!x} ,\ p \in P\ \} \cup \{\ \delta_k \mid p \xrightarrow{\delta_k} p,\ p \in P\ \}$

Testing Transition Systems: Extensions

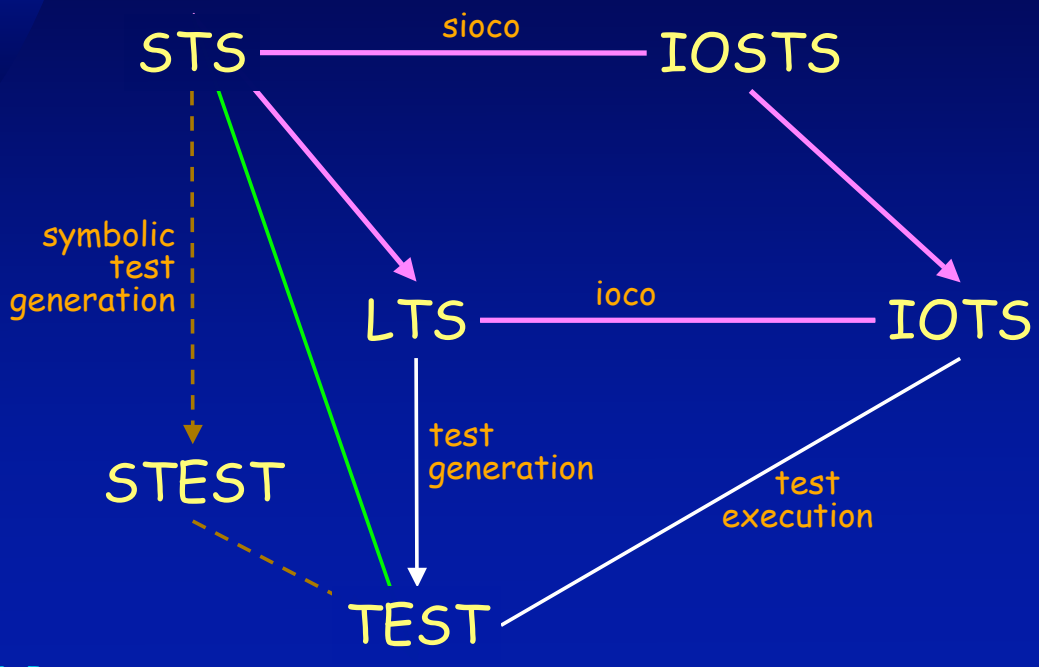


Transition System with Data



Disadvantages:
- infinity
- loss of information
  (e.g. for test selection)

Symbolic Data



STS $\mathcal{S} = \langle L, l_0, \mathcal{V}, \mathcal{I}, \Lambda, \rightarrow \rangle$, where

- L: set of *locations*
- $l_0 \in L$: *initial location*
- $\mathcal{V}$: set of *location variables*
- $\mathcal{I}$: set of *interaction variables*
- Λ: set of *gates*
- $\rightarrow \subseteq L \times \Lambda_\tau \times \mathcal{F}(Var) \times \mathcal{T}(Var)^{\mathcal{V}} \times L$: *switch relation*

and moreover

- $\mathcal{V} \cap \mathcal{I} = \emptyset$
- $Var =_{def} \mathcal{V} \cup \mathcal{I}$
- $\tau \notin \Lambda$: *unobservable gate*
- $l \xrightarrow{\lambda, \varphi, \rho} l' \;=\; (l, \lambda, \varphi, \rho, l') \in \rightarrow$, where
  - φ: *switch restriction* with $\mathsf{free}(\varphi) \subseteq \mathcal{V} \cup \mathsf{type}(\lambda)$
  - ρ: *update mapping* with $\rho \in \mathcal{T}(\mathcal{V} \cup \mathsf{type}(\lambda))^{\mathcal{V}}$
- $\mathsf{type}(\lambda)$: tuple of distinct interaction variables for gate λ
- $\mathsf{type}(\tau) = \langle\,\rangle$
- $\mathcal{S}(\iota)$: *initialised STS*, where $\iota \in \mathcal{U}^{\mathcal{V}}$ initialises all variables from $\mathcal{V}$ in l_0

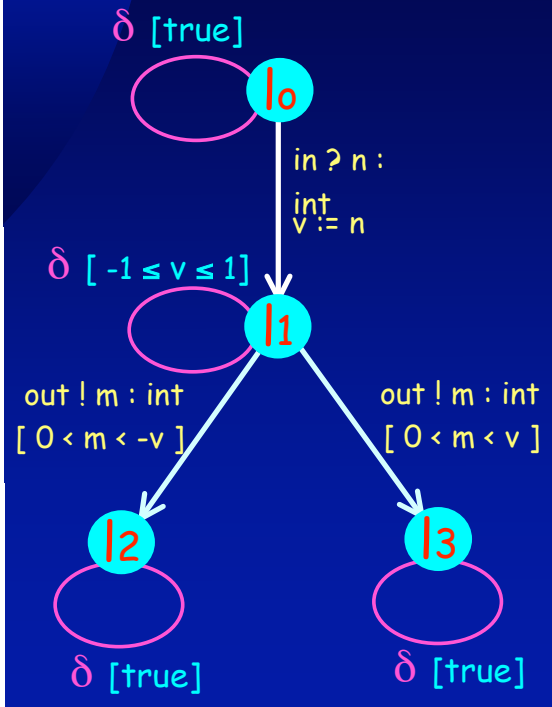
Symbolic Transition System

Semantics $[\![\mathcal{S}]\!]_\iota$ **of STS** $\mathcal{S} = \langle L, l_0, \mathcal{V}, \mathcal{I}, \Lambda, \rightarrow \rangle$
in the context of initialisation $\iota \in \mathcal{U}^{\mathcal{V}}$ is
LTS $\langle L \times \mathcal{U}^{\mathcal{V}}, (l_0, \iota), \Sigma, \rightarrow \rangle$, where

- $\Sigma = \bigcup_{\lambda \in \Lambda} (\{\lambda\} \times \mathcal{U}^{|\mathsf{type}(\lambda)|})$ is the set of actions
- $\rightarrow \subseteq (L \times \mathcal{U}^{\mathcal{V}}) \times (\Sigma \cup \{\tau\}) \times (L \times \mathcal{U}^{\mathcal{V}})$ is defined by:

$$\frac{l \xrightarrow{\lambda, \varphi, \rho} l' \quad \varsigma \in \mathcal{U}^{\mathsf{type}(\lambda)} \quad \vartheta \cup \varsigma \models \varphi \quad \vartheta' = (\vartheta \cup \varsigma)_{\mathsf{eval}} \circ \rho}{(l, \vartheta) \xrightarrow{(\lambda, \varsigma(\mathsf{type}(\lambda)))} (l', \vartheta')}$$

Symbolic Quiescence



Symbolic quiescence in location l_1:

$$\Delta(l_1) = \neg\exists\ m{:}int\ .\ 0 < m < -v \wedge \neg\exists\ m{:}int\ .\ 0 < m < v = -1 \le v \le 1$$

Symbolic suspension switch relation

$$l_0 \xRightarrow{in?\ \delta\ \ [-1 \le n_1 \le 1]\ \ v:=n_1} l_1$$

Symbolic state

$(l_1, [-1 < n_1 < 1], v:=n_1)$

Semantics of symbolic state

$\{ (l_1, -1), (l_1, 0), (l_1, 1) \}$

Symbolic Trace, After, . . .

Symbolic suspension trace

…… pair of …… (sequence of gates,
formula over indexed interaction
variables and location variables) ……

Symbolic $after_s$

…… <symbolic state> $after_s$ <symbolic suspension trace> ……

Lemma

[[<symbolic state> $after_s$ <symbolic suspension trace>]]
=
[[<symbolic state>]] after [[<symbolic suspension trace>]]

Symbolic ioco

Specification: IOSTS $\mathcal{S}(\iota_S) = \langle L_S, l_S, \mathcal{V}_S, \mathcal{I}, \Lambda, \rightarrow_S \rangle$
Implementation: IOSTS $\mathcal{P}(\iota_P) = \langle L_P, l_P, \mathcal{V}_P, \mathcal{I}, \Lambda, \rightarrow_P \rangle$
both initialised, implementation input-enabled, $\mathcal{V}_S \cap \mathcal{V}_P = \emptyset$
$\mathcal{F}_s$: a set of symbolic extended traces satisfying $[\![\mathcal{F}_s]\!]_{\iota_S} \subseteq Straces((l_0, \iota))$;

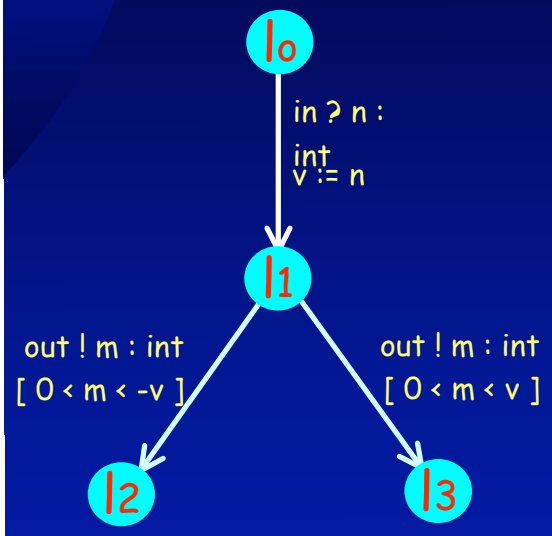
$\mathcal{P}(\iota_P)$ $\mathbf{sioco}_{\mathcal{F}_s}$ $\mathcal{S}(\iota_S)$ iff

$\forall (\sigma, \chi) \in \mathcal{F}_s \ \forall \lambda_\delta \in \Lambda_U \cup \{\delta\} : \iota_P \cup \iota_S \models \overline{\forall}_{\widehat{\mathcal{I}} \cup \mathcal{I}} (\Phi(l_P, \lambda_\delta, \sigma) \wedge \chi \rightarrow \Phi(l_S, \lambda_\delta, \sigma))$
where $\Phi(\xi, \lambda_\delta, \sigma) = \bigvee \{\varphi \wedge \psi \mid (\lambda_\delta, \varphi, \psi) \in \mathbf{out}_s((\xi, \top, \mathsf{id})_0 \mathbf{after}_s(\sigma, \top))\}$

Theorem 1.

$\mathcal{P}(\iota_P)$ $\mathbf{sioco}_{\mathcal{F}_s}$ $\mathcal{S}(\iota_S)$ *iff* $[\![\mathcal{P}]\!]_{\iota_P}$ $\mathbf{ioco}_{[\![\mathcal{F}_s]\!]_{\iota_S}}$ $[\![\mathcal{S}]\!]_{\iota_S}$

An Application: Coverage



Location coverage
l_0, l_1, l_2, l_3

Semantic state coverage
l_0, $(l_1,0)$, $(l_1,1)$, $(l_1,2)$, $(l_1,3)$, ……,
$(l_1,-1)$, $(l_1,-2)$, $(l_1,-3)$, ……,
$(l_2,2)$,……, $(l_2,2)$, ……

Symbolic state coverage
(l_0, [true], v:=v)
(l_1, [true], v:=n_1)
(l_1, [-1 < n_1 < 1], v:=n_1)
(l_2, [0 < m_2 < -n_1], v:=n_1)
(l_3, [0 < m_2 < n_1], v:=n_1)

Concluding

- Testing can be formal, too (M.-C. Gaudel, TACAS'95)
  - Testing shall be formal, too
- A test generation algorithm is not just another algorithm :
  - Proof of soundness and exhaustiveness
  - Definition of test assumption and implementation relation
- For labelled transition systems :
  - ioco for expressing conformance between imp and spec
  - a sound and exhaustive test generation algorithm
  - tools generating and executing tests:
    TGV, TestGen, Agedis, TorX, ...

Perspectives

Model based formal testing can improve the testing process :

- ☺ model is precise and unambiguous basis for testing
  - design errors found during validation of model
- ☺ longer, cheaper, more flexible, and provably correct tests
  - easier test maintenance and regression testing
- ☺ automatic test generation and execution
  - full automation : test generation + execution + analysis
- ☺ extra effort of modelling compensated by better tests

Thank You

Chapter 10 : Program Verification by Invariant Technique

1. Expression, substitution, proper state
2. Semantics and proof of a program
3. Proof system

Verification of sequential programs by invariant techniques

Note

These slides are a summary of the first part of the course :
“Preuves automatiques et preuves de programmes”
given till 2007 by Prof. Jean-François Raskin

References

References

- Verification of sequential and concurrent programs, K. R. Apt, E.-R. Olderog, Springer-Verlag, 1991.
- Logic in Computer Science, Modeling and Reasoning about Systems, R. A. Huth, M. D. Ryan, Cambridge University Press, 1999.
- The B-Book, Assigning Programs to Meanings, Cambridge University Press, 1995.
- Program Verification, Nissim Francez, Addison-Wesley Plublishing Company, 1992.

Plan

Variables, constants, expressions

Variables, constants, expressions

- Simple variables and arrays
- simple constants
- relations and functions (= higher order constants)
- **if** B **then** s_1 **else** s_2

Expressions s and Assertions p

- $Var(s)$: set of variables of s
- $Free(p)$: set of variables not bounded by a quantifier $\exists x$ or $\forall x$

Substitution $s \ll u := t \gg$

Unformally it gives the expresion s where the variable u is inductively replaced by the expression t

Substitution $s \ll u := t \gg$

- A formal definition of substitution can be defined (not given here).
- When s contains arrays or quantifiers, the definition needs some care.

Semantic of expressions and assertions

Semantics of expressions and assertions

- The semantic value of an expression s is an element in a semantic domain.
- Notation : $\mathcal{I}[\![s]\!]$: value in the semantic domain of s.
- $\mathcal{D}_T$ domain of a type T.
- $\mathcal{D}_{\textbf{Integer}} = \{0, 1, -1, 2, -2, \ldots\}$;
- $\mathcal{D}_{\textbf{Boolean}} = \{\textbf{true}, \textbf{false}\}$;
- $\mathcal{D}_{T_1 \times \cdots \times T_n \Rightarrow T} = \mathcal{D}_{T_1} \times \cdots \times \mathcal{D}_{T_n} \Rightarrow \mathcal{D}_T$,
- The semantic domain

$$\mathcal{D} = \bigcup_T \text{ a type } \mathcal{D}_T$$

Interpretation of variables

Interpretation of variables

The semantics of variables is not fixed ; but it is given with the notion of proper state :

$$\sigma : Var \to \mathcal{D}$$

with $\sigma(x) \in \mathcal{D}_T$ if x is of type T.

Then if a is an array of n dimensions $\sigma(a)$ is a function of type $\mathcal{D}_{T_1} \times \cdots \times \mathcal{D}_{T_n} \to \mathcal{D}_T$, and if $d_1 \in \mathcal{D}_{T_1}, \ldots, d_n \in \mathcal{D}_{T_n}$ then $\sigma(a)(d_1, \ldots, d_n) \in \mathcal{D}_T$.

Semantics of expressions in a proper state

$\mathcal{I}[\![s]\!] : \Sigma \to D$

- If s is a simple variable :

$$\mathcal{I}[\![s]\!](\sigma) = \sigma(s)$$

- If s is a constant of a basic type which denotes value d :

$$\mathcal{I}[\![s]\!](\sigma) = d$$

- If $s \equiv op(s_1, \ldots, s_n)$ with the constant op of higher type which denotes the function f :

$$\mathcal{I}[\![s]\!](\sigma) = f(\mathcal{I}[\![s_1]\!](\sigma), \ldots, \mathcal{I}[\![s_n]\!](\sigma))$$

Semantics of expressions in a proper state (cont'd)

$\mathcal{I}[\![s]\!] : \Sigma \to D$

- $s \equiv a[s_1, \ldots, s_n]$ is an array :

$$\mathcal{I}[\![s]\!](\sigma) = \sigma(a)[\mathcal{I}[\![s_1]\!](\sigma), \ldots, \mathcal{I}[\![s_n]\!](\sigma)]$$

- $s \equiv$ **if** B **then** s_1 **else** s_2 :

$$\mathcal{I}[\![s]\!](\sigma) = \begin{cases} \mathcal{I}[\![s_1]\!](\sigma) & \text{if } \mathcal{I}[\![B]\!](\sigma) = \textbf{true}; \\ \mathcal{I}[\![s_2]\!](\sigma) & \text{if } \mathcal{I}[\![B]\!](\sigma) = \textbf{false}. \end{cases}$$

- $s \equiv (s_1)$:

$$\mathcal{I}[\![s]\!](\sigma) = \mathcal{I}[\![s_1]\!](\sigma)$$

$\mathcal{I}[\![\cdot]\!]$ is fixed for the constants : hence $\mathcal{I}[\![s]\!](\sigma)$ is shorten by $\sigma(s)$.

Semantics of expressions and assertions : update of a proper state

$\sigma \ll u := d \gg$

where u is a simple or indexed variable of type T.

- If u is a simple variable : $\sigma \ll u := d \gg$ is the proper state where u has the value d and the value of the other variables is the same than the one in σ ;
- If $u \equiv a[t_1, \ldots, t_n]$ then $\sigma \ll u := d \gg$ is the proper state which gives the same value than σ to all variables except for a :

$$\sigma \ll u := d \gg (a)(d_1, \ldots, d_n) =$$

$$\begin{cases} d & \text{if } \bigwedge_i d_i = \sigma(t_i) \\ \sigma(a)(d_1, \ldots, d_n) & \text{otherwise.} \end{cases}$$

Semantics of expressions and assertions (cont'd) : states defined by an assertion p

$[\![p]\!] = \{\sigma \in \Sigma | \sigma \models p\}$

Some properties :

- $[\![\neg p]\!] = \Sigma \setminus [\![p]\!]$;
- $[\![p \vee q]\!] = [\![p]\!] \cup [\![q]\!]$;
- $[\![p \wedge q]\!] = [\![p]\!] \cap [\![q]\!]$;
- $p \rightarrow q \iff [\![p]\!] \subseteq [\![q]\!]$;
- $p \leftrightarrow q \iff [\![p]\!] = [\![q]\!]$.

Substitution lemma

The restriction of a proper state σ to a subset of variables X, is denoted :

$$\sigma\lfloor X \rfloor$$

lemma

For any assertion p, expressions s, r and proper states σ and τ :

- If $\sigma\lfloor Var(s)\rfloor = \tau\lfloor Var(s)\rfloor$ then $\sigma(s) = \tau(s)$;
- If $\sigma\lfloor Free(p)\rfloor = \tau\lfloor Free(p)\rfloor$ then $\sigma \models p$ iff $\tau \models p$.

Substitution lemma

For any assertion p, expressions s and t, u a simple or indexed variable of same type than t, and a proper state σ :

1. $\sigma(s \ll u := t \gg) = \sigma \ll u := \sigma(t) \gg (s)$;
2. $\sigma \models p \ll u := t \gg \iff \sigma \ll u := \sigma(t) \gg \models p$

The proofs are omitted here

Plan

Formal proof

Hoare triples

Correctness formula

$$\{p\}\ P\ \{q\}$$

Proof system :

- Axioms : “given formulas” ;
- Proof rules : used to establish “new” formulas from axioms or already established formulas.

Formal proof

Format of a rule

$$\frac{\phi_1, \ldots, \phi_n}{\psi} \text{ where "..."}$$

This rule says that ψ can be established if $\phi_1, \ldots, \phi_n$ have already been established and if "..." is verified.

Definitions

- A proof is a sequence of formulas $\varphi_1, \ldots, \varphi_n$ such that φ_i is an axiom or can be established from formulas in $\{\varphi_1, \ldots, \varphi_{i-1}\}$ with proof rules.
- A theorem is the last formula of a proof.
- Given a proof system P, we denote $\vdash_P \psi$ if ψ can be established from the proof system P.

A programming language and its formal semantics

Syntax

$$\begin{array}{rl} S ::= & \mathit{skip} \\ & \mid u := t \\ & \mid S_1 ; S_2 \\ & \mid \textbf{if } B \textbf{ then } S_1 \textbf{ else } S_2 \textbf{ fi} \\ & \mid \textbf{while } B \textbf{ do } S_1 \textbf{ od} \end{array}$$

u is a simple or index variable, t is an expression and B is a boolean expression.

We suppose programs are well typed.

Abbrevation :

if B **then** S_1 **fi**
is the short for
if B **then** S_1 **else** *skip* **fi**

A programming language and its formal semantics

Semantics of a program

A program defines a function from initial states to final states :

$$\mathcal{M}[\![S]\!] : \Sigma \to \Sigma \cup \{\bot\}$$

where $\bot$ denotes divergence.

Two approaches exist to define $\mathcal{M}[\![S]\!]$:

- the denotational and
- the operational approach.

A programming language and its formal semantics

We suppose a high level operational semantics where assignments and tests are atomic.

transitions between “configurations”

$$\langle S, \sigma \rangle \to \langle R, \tau \rangle$$

Execute S from the state σ produces the state τ and R is the part of the program which remains to be excuted.

Note : to express the termination, we can have $R \equiv E$ (the empty program).

$\mathcal{M}[\![S]\!]$ is based on the relation $\to$ on S.

The relation $\rightarrow$ can be defined with a formal proof system (Hennessy and Plotkin), the result is a transition system.

Proof system for a deterministic and sequential program

It is defined with the following axioms and inference rules :

1. $\langle skip, \sigma \rangle \rightarrow \langle E, \sigma \rangle$
2. $\langle x := t, \sigma \rangle \rightarrow \langle E, \sigma \ll x := \sigma(t) \gg \rangle$
3. $\dfrac{\langle S_1, \sigma \rangle \rightarrow \langle S_2, \tau \rangle}{\langle S_1; S, \sigma \rangle \rightarrow \langle S_2; S, \tau \rangle}$ (and $E; S \equiv S$)
4. $\langle \textbf{if } B \textbf{ then } S_1 \textbf{ else } S_2 \textbf{ fi}, \sigma \rangle \rightarrow \langle S_1, \sigma \rangle$ where $\sigma \models B$
5. $\langle \textbf{if } B \textbf{ then } S_1 \textbf{ else } S_2 \textbf{ fi}, \sigma \rangle \rightarrow \langle S_2, \sigma \rangle$ where $\sigma \not\models B$
6. $\langle \textbf{while } B \textbf{ do } S_1 \textbf{ od}, \sigma \rangle \rightarrow \langle S_1; \textbf{while } B \textbf{ do } S_1 \textbf{ od}, \sigma \rangle$ where $\sigma \models B$
7. $\langle \textbf{while } B \textbf{ do } S_1 \textbf{ od}, \sigma \rangle \rightarrow \langle E, \sigma \rangle$ where $\sigma \not\models B$

$\langle S, \sigma \rangle \rightarrow \langle R, \tau \rangle$ is possible iff it can be deduced with the proof system.

A programming language and its formal semantics

Definitions

- A transitions sequence of S which starts in σ is a finite or infinite sequence of configurations $\langle S_i, \sigma_i \rangle$ such that
$$\langle S, \sigma \rangle \rightarrow \langle S_1, \sigma_1 \rangle \rightarrow \cdots \rightarrow \langle S_i, \sigma_i \rangle \rightarrow \ldots$$
- An execution of S which starts in σ is a transition sequence from σ which cannot be extended.
- An execution of S which starts in σ is divergent if the execution is infinite.

We consider as programs :

- deterministic : for each pairs $\langle S, \sigma \rangle$ there is at most one successor for $\rightarrow$;
- non blocking : if $S \not\equiv E$ then each $\sigma \in \Sigma$, $\langle S, \sigma \rangle$ has a successor for $\rightarrow$

A programming language and its formal semantics

Semantics of S

We consider two semantics :

- partial correctness semantics :
$$\mathcal{M}[\![S]\!](\sigma) = \{\tau | \langle S, \sigma\rangle \rightarrow^* \langle E, \tau\rangle\}$$
- total correctness semantics :
$$\mathcal{M}_{tot}[\![S]\!](\sigma) = \{\tau | \langle S, \sigma\rangle \rightarrow^* \langle E, \tau\rangle\} \cup \{\perp | S \textit{ diverges from } \sigma\}$$

A programming language and its formal semantics

Some more notions :

- $\Omega \equiv$ **while** *true* **do** *skip* **od**, a never ending program ;
- (**while** B **do** S **od**)$^0 = \Omega$;
- (**while** B **do** S **od**)$^{k+1}$
$= $ **if** B **then** S; (**while** B **do** S **od**)^k **else** *skip* .

A programming language and its formal semantics

Semantic function

- $\mathcal{M}[\![S]\!]$, $\mathcal{M}_{tot}[\![S]\!]$ are extended to $\Sigma \cup \{\bot\}$ with :
$$\mathcal{M}[\![S]\!](\bot) = \varnothing \text{ and } \mathcal{M}_{tot}[\![S]\!](\bot) = \{\bot\}$$
- and to sets with :
$$\mathcal{M}[\![S]\!](X) = \bigcup_{\sigma \in X} \mathcal{M}[\![S]\!](\sigma)$$
$$\mathcal{M}_{tot}[\![S]\!](X) = \bigcup_{\sigma \in X} \mathcal{M}_{tot}[\![S]\!](\sigma)$$

  Notation : $\mathcal{N}[\![S]\!]$ stands for $\mathcal{M}[\![S]\!]$ and $\mathcal{M}_{tot}[\![S]\!]$

A programming language and its formal semantics

Some properties of semantic functions

- $\mathcal{N}[\![S]\!]$ is monotone ;
- $\mathcal{N}[\![S_1 ; S_2]\!](X) = \mathcal{N}[\![S_2]\!](\mathcal{N}[\![S_1]\!](X))$;
- $\mathcal{N}[\![(S_1 ; S_2) ; S_3]\!](X) = \mathcal{N}[\![S_1 ; (S_2 ; S_3)]\!](X)$;
- $\mathcal{N}[\![\textbf{if } B \textbf{ then } S_1 \textbf{ else } S_2 \textbf{fi}]\!](X) =$
$\mathcal{N}[\![S_1]\!](X \cap [\![B]\!]) \cup \mathcal{N}[\![S_2]\!](X \cap [\![\neg B]\!])$;
- $\mathcal{M}[\![\textbf{while } B \textbf{ do } S_1 \textbf{od}]\!]$
$= \bigcup_{k=0}^{\infty} \mathcal{M}[\![(\textbf{while } B \textbf{ do } S_1 \textbf{od})^k]\!]$

A programming language and its formal semantics

Definitions

For a program S

- $Var(S)$: variables used by S
- $Change(S)$: variables changed by S

Other properties

- For any proper states σ and τ, if $\tau \in \mathcal{N}[\![S]\!](\sigma)$:
$$\tau\lfloor Var \setminus Change(S)\rfloor = \sigma\lfloor Var \setminus Change(S)\rfloor$$
- for any proper states σ and τ such that $\tau\lfloor Var(S)\rfloor = \sigma\lfloor Var(S)\rfloor$:
$$\mathcal{N}[\![S]\!](\sigma)\lfloor Var(S)\rfloor = \mathcal{N}[\![S]\!](\tau)\lfloor Var(S)\rfloor$$

Definition of correct program

The correctness formula $\{p\}S\{q\}$ is true :

- for the partial correctness, denoted $\models \{p\}S\{q\}$, iff
$$\mathcal{M}[\![S]\!]([\![p]\!]) \subseteq [\![q]\!]$$
- for the total correctness, denoted $\models_{tot} \{p\}S\{q\}$, iff
$$\mathcal{M}_{tot}[\![S]\!]([\![p]\!]) \subseteq [\![q]\!]$$

Plan

Proof system for the partial correctness

Fact

The previous method involves semantic definitions which are not obvious to check.
A formal proof system can be used which directly uses correctness formulae.

Proof system for the partial correctness

Axioms and rules of the proof system for partial correctness :

- skip Ax1 : $\{p\}skip\{p\}$
- assignment Ax2 : $\{p \ll u := t \gg\}u := t\{p\}$
- Composition R3 : $\dfrac{\{p\}S_1\{r\}, \{r\}S_2\{q\}}{\{p\}S_1; S_2\{q\}}$
- Test R4 : $\dfrac{\{p \wedge B\}S_1\{q\}, \{p \wedge \neg B\}S_2\{q\}}{\{p\}\textbf{if } B \textbf{ then } S_1 \textbf{ else } S_2\textbf{fi}\{q\}}$
- Iteration R5 : $\dfrac{\{p \wedge B\}S\{p\}}{\{p\}\textbf{while } B \textbf{ do } S \textbf{ od}\{p \wedge \neg B\}}$
- consequence R6 : $\dfrac{p \rightarrow p_1, \{p_1\}S\{q_1\}, q_1 \rightarrow q}{\{p\}S\{q\}}$

Proof system for the partial correctness

Example of partail correctness

For

$$S \equiv \begin{array}{l} x := 1; \\ a[x] := 2; \end{array}$$

do we have ?

$$\models^{?} \{\textbf{true}\}S\{a[1] = 2\}$$

Proof system for the partial correctness

Example from Tony Hoare : the integer division

- specification :

$$\{x \geq 0 \wedge y \geq 0\}$$
$$\texttt{DIV}$$
$$\{quo \cdot y + rem = x \wedge 0 \leq rem < y\}$$

- the program `DIV` :

quo := 0;
rem := *x*;
while *rem* ≥ *y* **do**
  rem := *rem* − *y*;
  quo := *quo* + 1;
end while

Proof system for the partial correctness

To establish :

$$\{x \geq 0 \wedge y \geq 0\}$$
$$\texttt{DIV}$$
$$\{quo \cdot y + rem = x \wedge 0 \leq rem < y\}$$

we have to find an invariant p such that :

1. The invariant is verified when the while is reached :
   $\{x \geq 0 \wedge y \geq 0\}$
   $quo := 0; rem := x;$
   $\{p\}$
2. The invariant remains true after each iteration :
   $\{p \wedge B\}$
   $rem := rem - y; quo := quo + 1$
   $\{p\}$
3. When the condition in the while becomes false it implies the post-condition :
   $p \wedge \neg B \rightarrow quo \cdot y + rem = x \wedge 0 \leq rem < y$

Find $p : p \equiv quo \cdot y + rem = x \wedge rem \geq 0$

Generally the invariant is found by weakening the post-condition (it cannot be automatized).

Proof system for total correctness

Axioms and rules for the total correctness

Axioms and rules for the partial correctness (A1-R6), together with :

- iteration II R7 :

$$\frac{\{p \wedge B\}S\{p\}, \quad \{p \wedge B \wedge t = z\}S\{t < z\}, \quad p \rightarrow t \geq 0}{\{p\}\textbf{while } B \textbf{ do } S_1 \textbf{ od}\{p \wedge \neg B\}}$$

t is called termination function.

Total correctness Example : the integer division

Termination function ?

$$t \equiv rem$$

The invariant used for the partial correctness was too weak.
We need the following invariant for the proof of the total correctness :

$$p' \equiv quo \cdot y + rem = x \wedge rem \geq 0 \wedge y > 0$$

Partial and total correctness

Full example : Fibonacci's suite

Definition of Fibonacci's suite :

$F_0 = 0$
$F_1 = 1$
$F_n = F_{n-1} + F_{n-2}$, with $n \geq 2$

Notation $fib(n) = F_n$
Specification :
$\{n \geq 0\} S \{x = fib(n)\}$

Program $S \equiv$
$x := 0;$
$y := 1;$
count $:= n;$
while count>0 **do**
$h := y;$
$y := x + y;$
$x := h;$
$count := count - 1$
end while

Partial and total correctness : Fibonacci's suite

Which intermediate formulas do we need to establish ?

1. $\vdash^? \{n \geq 0\} x := 0; y := 0; count := n; \{p\}$;
   The invariant p is true when the while is reached :
2. $\vdash^? \{p \wedge count > 0\} h := y; y := x + y; x := h; count := count - 1; \{p\}$;
   The loops preserves the invariant ;
3. $\vdash^? p \wedge \neg(count > 0) \rightarrow x = fib(n)$.
   At the exit of the while, the postcondition is verified ;

Partial and total correctness : Fibonacci's suite

For the total correctness :

1. $\vdash^? \{t = z\} h := y; y := x + y; x := h; count := count - 1; \{t < z\}$
   The termination function t decreases at each loop's iteration
2. $\vdash^? p \rightarrow t \geq 0$;
   The termination function has always a positive value when the invariant is true.

Partial and total correctness : Fibonacci's suite

Which invariant do we need ?

$$\begin{aligned} p \equiv x = \quad & fib(n - count) \\ & \wedge y = fib(n - count + 1) \\ & \wedge count \geq 0 \end{aligned}$$

Which termination function do we need ?

$$t \equiv count$$

Soundness and completeness of the proof systems

When using the proof systems $\vdash$ and $\vdash_{\text{Tot}}$, we directly use correctness formulas.
Are we sure that the proofs in $\vdash$ and $\vdash_{\text{Tot}}$ really means that the program is correct ?
Two questions on $\vdash$ and $\vdash_{\text{Tot}}$ are important :

- Are they sound : are the proven correctness formulas established from $\vdash$ and $\vdash_{\text{Tot}}$ valid ?
- Are they complete : for any correct program, can we use these proof systems to establish correctness ?

Soundness and completeness of the proof systems

Recalls

- Semantic definition of partial correctness :

$$\models \{p\}S\{q\}$$
$$\textbf{iff } \forall \sigma \in [\![p]\!] : \mathcal{M}[\![s]\!](\sigma) = \varnothing \vee \mathcal{M}[\![s]\!](\sigma) \in [\![q]\!] \textbf{ iff } \mathcal{M}[\![s]\!]([\![p]\!]) \subseteq [\![q]\!]$$

- Syntaxical definition of partial correctness :

$$\vdash \{p\}S\{q\}$$
iff there exists a theorem in the proof system (corresponding to the partial correctness semantics) for the correctness formula.

Soundness of the proof systems

Definition : sound proof system

The proof system $\vdash$ is sound for $\models$ iff :

$$\vdash \phi \text{ implies } \models \phi, \text{ for all formula } \phi$$

In our case, we want to establish that :

- $\vdash$ is sound for the partial correctness :
$$\vdash \{p\}S\{q\} \text{ implies } \models \{p\}S\{q\}$$
- $\vdash_{\text{Tot}}$ is sound for the total correctness :
$$\vdash_{\text{Tot}} \{p\}S\{q\} \text{ implies } \models_{\text{Tot}} \{p\}S\{q\}$$

Completeness of the proof systems

Recall

A proof system is complete if it allows to establish the proof of any valid formula, i.e. :

$$\models \phi \text{ implies } \vdash \phi, \text{ for any formula } \phi$$

In our case we want to establish that :

- for all partial correctness formula such that : $\models \{p\}S\{q\}$, we have $\vdash \{p\}S\{q\}$
- for all total correctness formula such that : $\models_{\text{Tot}} \{p\}S\{q\}$, we have $\vdash_{\text{Tot}} \{p\}S\{q\}$

Completeness of the proof systems : relative completeness

The notion of completeness that we study is relative to the assertion language used and to its interpretation.

Hypothesis needed

The proof system is extended with all the valid assertions.
For instance we will have :

- $\models 3 \times (a+b) = 3 \times a + 3 \times b$ and then $\vdash 3 \times (a+b) = 3 \times a + 3 \times b$;
- $\models a \geq b \rightarrow a \times a \geq b \times b$ and then $\vdash a \geq b \rightarrow a \times a \geq b \times b$;
- ...

Completeness of the proof systems : Weakest precondition (Dijkstra)

Given the deterministic sequential program S and Φ a set of proper states.

Definitions : weakest (liberal) precondition

- $\text{wlp}(S, \Phi) = \{\sigma \mid \mathcal{M}[\![S]\!](\sigma) \subseteq \Phi\}$
- $\text{wp}(S, \Phi) = \{\sigma \mid \mathcal{M}_{tot}[\![S]\!](\sigma) \subseteq \Phi\}$

- wlp = weakest liberal precondition
- wp = weakest precondition

Completeness of the proof systems : Weakest precondition (Dijkstra)

wlp(S, Φ) / wp(S, Φ)

- wlp(S, Φ) is the set of proper states from which if S is executed , and if the execution terminates then the final state belongs to Φ ;
- wp(S, Φ) is the set of proper states from which if S is executed , the execution does terminate and the final state belongs to Φ.

Completeness of the proof systems / completeness of the assertion language

Lemma

For any program S and assertion q,

1. there exists an assertion p such that
$$[\![p]\!] = \text{wlp}(S, [\![q]\!])$$
2. there exists an assertion p such that
$$[\![p]\!] = \text{wp}(S, [\![q]\!])$$

Completeness of the proof systems

Properties of the operators wlp and wp

For any program S, S_1, S_2 and assertions p and q :

1. $\models \text{wlp}(skip, q) \leftrightarrow q$
2. $\models \text{wlp}(u := t, q) \leftrightarrow q \ll u := t \gg$
3. $\models \text{wlp}(S_1; S_2, q) \leftrightarrow \text{wlp}(S_1, \text{wlp}(S_2, q))$
4. $\models \text{wlp}(\textbf{if } B \textbf{ then } S_1 \textbf{ else } S_2, q) \leftrightarrow (\text{wlp}(S_1, q) \wedge B) \vee (\text{wlp}(S_2, q) \wedge \neg B)$
5. $\models \text{wlp}(S, q) \wedge B \leftrightarrow \text{wlp}(S_1, \text{wlp}(S, q))$, where $S \equiv \textbf{while } B \textbf{ do } S_1 \textbf{ od}$
6. $\models \text{wlp}(S, q) \wedge \neg B \leftrightarrow q$ where $S \equiv \textbf{while } B \textbf{ do } S_1 \textbf{ od}$
7. $\models \{p\}S\{q\}$ iff $p \rightarrow \text{wlp}(S, q)$

There properties are also true with the operator wp.

Completeness of the proof systems : completeness of the expressions language

Second hypothesis needed

We consider that the language of expressions is expressive in the following way :
Lemma : for any computable partial function : $F : \Sigma \rightarrow$ **integer**, there is an integer expression t such that for any proper state σ, if $F(\sigma)$ is defined then :

$$F(\sigma) = \sigma(t)$$

Completeness of the proof systems

Completeness theorem

The proof systems $\vdash$ and $\vdash_{\text{Tot}}$ are complete for the partial and total correctness.

Proof omitted

References

References I

- Jean-Raymond Abrial, The B-book, Cambridge University Press, 1996.
- Dragan Bosnacki, Dennis Dams, and Leszek Holenderski, Symmetric spin., STTT **4** (2002), no. 1, 92–106.
- E. M. Clarke, R. Enders, T. Filkorn, and S. Jha, Exploiting symmetry in temporal logic model checking, Form. Methods Syst. Des. **9** (1996), no. 1-2, 77–104.
- David L. Dill, Andreas J. Drexler, Alan J. Hu, and C. Han Yang, Protocol verification as a hardware design aid, International Conference on Computer Design, 1992, pp. 522–525.
- Alastair F. Donaldson, Alice Miller, and Muffy Calder, Spin-to-grape : A tool for analysing symmetry in promela models., Electr. Notes Theor. Comput. Sci. **139** (2005), no. 1, 3–23.

References II

- E. Allen Emerson and A. Prasad Sistla, Utilizing symmetry when model checking under fairness assumptions : An automata-theoretic approach., CAV (Pierre Wolper, ed.), Lecture Notes in Computer Science, vol. 939, Springer, 1995, pp. 309–324.
- E. Allen Emerson and A. Prasad Sistla, Symmetry and model checking, Formal Methods in System Design **9** (1996), no. 1/2, 105–131.
- Martijn Hendriks, Gerd Behrmann, Kim Guldstrand Larsen, Peter Niebert, and Frits W. Vaandrager, Adding symmetry reduction to uppaal., FORMATS (Kim Guldstrand Larsen and Peter Niebert, eds.), Lecture Notes in Computer Science, vol. 2791, Springer, 2003, pp. 46–59.
- Gerard J. Holzmann, An improved protocol reachability analysis technique., Softw., Pract. Exper. **18** (1988), no. 2, 137–161.
- C. Norris Ip and David L. Dill, Better verification through symmetry., Formal Methods in System Design **9** (1996), no. 1/2, 41–75.

References III


Somesh Jha, Symmetry and induction in model checking, Ph.D. thesis, School of Computer Science, Carnegie Mellon University, October 1996.


Michael Leuschel, Michael Butler, Corinna Spermann, and Edd Turner, Symmetry reduction for b by permutation flooding, B'2007, the 7th Int. B Conference - Tool Session (Besancon, France), LNCS, vol. 4355, Springer, January 2007, To appear.


A. Prasad Sistla, Viktor Gyuris, and E. Allen Emerson, Smc : a symmetry-based model checker for verification of safety and liveness properties., ACM Trans. Softw. Eng. Methodol. **9** (2000), no. 2, 133–166.