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

- Introduction .......................................................... 5
- Kripke Structures and Labeled Transition Systems ............. 28
- Temporal logics ......................................................... 68
- $\omega$-automata ......................................................... 108
- $\mu$-calculus ........................................................... 133
- Model Checking ......................................................... 156
- Symbolic and efficient Model Checking ............................. 172
- Specification Languages and Formal Description Techniques .... 285
- Testing ................................................................. 315
- Program Verification by Invariant Technique ...................... 364

Main References

Chapter 1: Introduction

1. Motivation
2. Formal verification
3. Brief historical facts on specification and verification
Motivation

Formal verification

Brief historical facts on specification and verification

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

**Weaknesses of embedded systems**
- Discrete systems ⇒ very sensitive,
- Complex ⇒ difficult to design,
- Embedded ⇒ difficult to monitor,
- Safety critical ⇒ must be correct.
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:
  ```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:
  ```c
  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
Motivation

Formal verification

Brief historical facts on specification and verification

Plan

1 Motivation

2 Formal verification

3 Brief historical facts on specification and 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

- 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

- **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

- 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.
Motivation

Formal verification

Brief historical facts on specification and verification

Plan

1 Motivation

2 Formal verification

3 Brief historical facts on specification and verification

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 $\omega$-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
Motivation

Formal verification

Brief historical facts on specification and verification

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;
- 1980: Edmund Clarke and Ellen Emerson defined the temporal logic CTL;
- 1982: Pratt and Kozen, \( \mu \)-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

1 Basic definitions
2 Equivalences, bisimulation and simulation relations
3 Quotients, products, predecessors and successors
4 Verification, Model Checking, Testing
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 : $s \xrightarrow{\mathcal{K}} s' \equiv (s, s') \in \mathcal{R}$)
- $\mathcal{L} : \mathcal{S} \rightarrow 2^\mathcal{V}$ the label function.
Multual exclusion
Peterson

\[ P_1 \quad \text{loop forever} \]
\[ : \]
\[ \langle b_1 := \text{true}; x := 2 \rangle; \quad (* \text{noncritical actions} *) \]
\[ \text{wait until} \ (x = 1 \lor \neg b_2) \quad (* \text{request} *) \]
\[ \text{do critical section od} \]
\[ b_1 := \text{false} \quad (* \text{release} *) \]
\[ : \]
\[ \text{end loop} \quad (* \text{noncritical actions} *) \]
Multual exclusion
Peterson

$PG_1$:

$noncrit_1$

$b_1 := false$

$wait_1$

$x = 1 \lor \neg b_2$

$crit_1$

$b_1 := true; x := 2$

$PG_2$:

$noncrit_1$

$b_2 := true; x := 1$

$wait_1$

$x = 2 \lor \neg b_1$

$crit_1$

Kripke Structure for Peterson

$\langle c_1, n_2, x = 2 \rangle$

$\langle n_1, n_2, x = 2 \rangle$

$\langle n_1, w_2, x = 2 \rangle$

$\langle c_1, w_2, x = 1 \rangle$

$\langle c_1, c_2, x = 1 \rangle$

$\langle n_1, w_2, x = 1 \rangle$

$\langle w_1, n_2, x = 2 \rangle$

$\langle w_1, w_2, x = 1 \rangle$

$\langle w_1, c_2, x = 2 \rangle$

$\langle w_1, w_2, x = 2 \rangle$
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**

A labeled transition system $\mathcal{M}$ is a tuple $\mathcal{M} = (I, S, R)$ with
- $S$: the finite set of states
- $I \subseteq S$: the set of initial states
- $R \subseteq S \times \mathcal{L} \times S$: the set of transitions

In the first chapters, we shall mainly work with Kripke structures.

---

**Railroad System**

- **Train**
  - far, approach, near
  - exit, enter
- **Controller**
  - approach, raise, lower, exit
  - 1, 2, 3
- **Gate**
  - up, raise, lower, down
Railroad System

Computation Path

Path of a Kripke Structure

- An infinite path of a Kripke structure $\mathcal{K} = (I, S, R, L)$ is a function $\pi : \mathbb{N} \rightarrow S$ with $\pi(0) \in I$ and $\forall t. \pi(t) \xrightarrow{\mathcal{K}} \pi(t+1)$

- A path can be seen as an infinite sequence of states $\pi = s_0 s_1 s_2 \ldots$ such that $s_0 \in I$ and $s_i \xrightarrow{\mathcal{K}} s_{i+1}$

- $Path_{\mathcal{K}}(s) = \{ \pi | \pi(0) = s \text{ and } \forall t. \pi(t) \xrightarrow{\mathcal{K}} \pi(t+1) \}$

- $Path_{\mathcal{K}}(S) = \bigcup_{s \in S} Path_{\mathcal{K}}(s)$

- Trace of a path $\pi$ : sequence $\lambda t. L(\pi(t))$

- Language $Lang(s)$ of a state $s$ : sequence $Lang(s) := \{ \lambda t. L(\pi(t)) | \pi \in Path_{\mathcal{K}}(s) \}$

- Language $Lang(\mathcal{K}) := \bigcup_{s \in I} Lang(s)$
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]
### Basic definitions

Equivalences, bisimulation and simulation relations
Quotients, products, predecessors and successors
Verification, Model Checking, Testing

## 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

### Obvious candidate: Isomorphism up to renaming of the states

#### Isomorphic structures

$\mathcal{K}_1 = (I_1, S_1, R_1, L_1)$ and $\mathcal{K}_2 = (I_2, S_2, R_2, L_2)$ are isomorphic, if there is a bijection $\Theta : S_1 \leftrightarrow S_2$ with

- $S_2 = \Theta(S_1)$
- $s_1 \in I_1 \iff \Theta(s_1) \in I_2$
- $s_1 \xrightarrow{\mathcal{K}_1} s_2 \iff \Theta(s_1) \xrightarrow{\mathcal{K}_1} \Theta(s_2)$
- $L_1(s_1) = L_2(\Theta(s_1))$

Generally much too strong equivalence!
**Simulation relation**

**Definition:**
Given $\mathcal{K}_1 = (I_1, S_1, R_1, L_1)$ and $\mathcal{K}_2 = (I_2, S_2, R_2, L_2)$, $\sigma \subseteq S_1 \times S_2$ is a simulation relation between $\mathcal{K}_1$ and $\mathcal{K}_2$ if the following holds:

1. $(s_1, s_2) \in \sigma$ implies $L_1(s_1) = L_2(s_2)$
2. $\forall (s_1, s_2) \in \sigma, \forall s'_1 : s_1 \xrightarrow{\sigma} s'_1, \exists s'_2 : s_2 \xrightarrow{\sigma} s'_2 \land (s'_1, s'_2) \in \sigma$
3. $\forall s_1 \in I_1, \exists s_2 \in I_2 : (s_1, s_2) \in \sigma$

**Diagram:**

![Diagram showing simulation relation](#)

**Illustration:**

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

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

**Proof:**

We show that $\forall \omega \in \text{Lang}(\mathcal{K}_1) \Rightarrow \omega \in \text{Lang}(\mathcal{K}_2)$

Let $\sigma$, the simulation relation in $S_1 \times S_2$ with $\forall s \in I_1, \exists s' \in I_2 : (s, s') \in \sigma$

Let $\pi$ the path in $\mathcal{K}_1$ with trace $\omega$

There is a corresponding path $\pi'$ in $\mathcal{K}_2$ with

$\forall i \in \mathbb{N}, (\pi_i, \pi'_i) \in \sigma$ (by induction)

Hence $\forall i \in \mathbb{N}, L(\pi_i) = L(\pi'_i)$

which concludes the proof.
Checking simulation preorder

Algorithm to check simulation

- \((s_1, s_2) \in \mathcal{H}_0 \iff L_1(s_1) = L_2(s_2)\)
- \((s_1, s_2) \in \mathcal{H}_{i+1} \iff (s_1, s_2) \in \mathcal{H}_i \land \forall s'_1 \in S_1, s_1 \xrightarrow{\kappa_1} s'_1 \exists s'_2 \in S_2. s_2 \xrightarrow{\kappa_2} s'_2 \land (s'_1, s'_2) \in \mathcal{H}_i\)
- Until stabilization \((\mathcal{H}_{i+1} = \mathcal{H}_i)\)

Computing the greatest simulation relation between \(\kappa_1\) and \(\kappa_2\)

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

\[\Rightarrow \text{no state related to } p_0 : \kappa_1 \not\mathcal{S} \kappa_2\]
Computing the greatest simulation between $K_2$ and $K_1$

$H_0 = \{ (q_0, p_0), (q_1, p_1), (q_2, p_2), (q_3, p_3), (q_4, p_3) \}$

$\Rightarrow$ stable and $(q_0, p_0) \in H : K_2 \preceq^S K_1$

Equivalence of Kripke structure / LTS

- Other obvious candidate: Language equivalence

Algorithm to test language inclusion ($K_1 \preceq^L K_2$)

1. Determinisation of $K_2 : K_2'$
2. Check if $K_1 \preceq^S K_2'$

$K_1 \preceq^L K_2 \iff K_1 \preceq^L K_2 \land K_2 \preceq^L K_1$

- But determinisation of Kripke structure / LTS (as finite automata) is hard
- computing simulation or bisimulation is more efficient
Bisimulation relation

Given $\mathcal{K}_1 = (I_1, S_1, R_1, L_1)$ and $\mathcal{K}_2 = (I_2, S_2, R_2, L_2)$, $\sigma \subseteq S_1 \times S_2$ is a bisimulation relation between $\mathcal{K}_1$ and $\mathcal{K}_2$ ($\mathcal{K}_1 \simeq_\sigma^B \mathcal{K}_2$) if the following holds:

1. $(s_1, s_2) \in \sigma$ implies $L_1(s_1) = L_2(s_2)$
2. $\forall (s_1, s_2) \in \sigma, \forall s'_1. s_1 \xrightarrow{\mathcal{K}_1} s'_1, \exists s'_2. s_2 \xrightarrow{\mathcal{K}_2} s'_2 \land (s'_1, s'_2) \in \sigma$
3. $\forall s_1 \in I_1, \exists s_2 \in I_2 : (s_1, s_2) \in \sigma$
4. $\forall s_2 \in I_2, \exists s_1 \in I_1 : (s_1, s_2) \in \sigma$

$\mathcal{K}_1 \simeq_\sigma^B \mathcal{K}_2$ if there exists a bisimulation $\sigma$ between $\mathcal{K}_1$ and $\mathcal{K}_2$

- $\mathcal{K}_1 \simeq_\sigma^B \mathcal{K}_2 \Rightarrow \mathcal{K}_1 \simeq^S \mathcal{K}_2$
- $\simeq^B$ 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

Basic definitions / notations

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

A partition \( \rho \) is compatible with the relation \( R \)

- \( \iff \forall i \in I, \forall x, y \in B_i, \mathcal{L}(x) = \mathcal{L}(y) \wedge \overline{R[x]} \cap B_j \neq \emptyset \iff R[y] \cap B_j \neq \emptyset \)
- \( \iff \forall i \in I, \forall x, y \in B_i, \mathcal{L}(x) = \mathcal{L}(y) \wedge \forall B, B' \in \rho, \text{ either } B' \subseteq R^{-1}[B], \text{ or } B' \cap R^{-1}[B] = \emptyset \)
\( \rho \) is compatible with the relation \( R \) : first criteria

A partition \( \rho \) is compatible with the relation \( R \) iff

\[
\forall i \in I, \forall x, y \in B_i, \mathcal{L}(x) = \mathcal{L}(y) \land R[x] \cap B_i \neq \emptyset \Leftrightarrow R[y] \cap B_i \neq \emptyset
\]

\( \rho \) is compatible with the relation \( R \) : second criteria

A partition \( \rho \) is compatible with the relation \( R \)

\[
\text{iff } \forall i \in I, \forall x, y \in B_i, \mathcal{L}(x) = \mathcal{L}(y) \land \forall B', B' \in \rho,
\]

\[
either B' \subseteq R^{-1}[B], \text{ or } B' \cap R^{-1}[B] = \emptyset
\]
Basic definitions
Equivalences, bisimulation and simulation relations
Quotients, products, predecessors and successors
Verification, Model Checking, Testing

Checking bisimulation

Proposition (link between bisimulation and compatibility)

\( \rho \) is a bisimulation iff \( \rho \) is compatible with \( R \).

Algorithm to compute the coarsest partition compatible with \( R \)

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

\[
\{ \\
r := \{B \mid \text{forall } x,y \in B. \text{Lang}(x) = \text{Lang}(y)\}; \quad // \text{classes} \\
W := r \quad // \text{splitter} \\
\text{while } W \text{ not empty do} \\
\{ \\
\text{select } B \text{ in } W; \\
\text{suppress } B \text{ in } W; \\
\text{call Interpred}(R^{-1}[B]); \quad // \text{Compute a set of } (B',X_1) \\
\quad // \text{with all classes } B' \text{ in } r \\
\quad // \text{incompatible with } R^{-1}[B] \\
\quad // X_1 \text{ is the resulting intersect.} \\
\}
\]

for all $(B',X_1)$ in interpred do
\[
\{ \\
X_2 = B' \setminus X_1; \\
\text{replace } B' \text{ by } X_1 \text{ and } X_2 \text{ in } r; \\
\text{if } B' \text{ in } W \text{ then} \\
\quad \text{replace } B' \text{ by } X_1 \text{ and } X_2 \text{ in } W; \\
\text{else} \\
\quad \text{add } X_1 \text{ and } X_2 \text{ in } W; \\
\text{fi} \\
\}
\]
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

Suppose all states have the same set of propositions

![Diagram of Paige-Tarjan's algorithm example]
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

1. Basic definitions
2. Equivalences, bisimulation and simulation relations
3. Quotients, products, predecessors and successors
4. Verification, Model Checking, Testing

Quotient structures

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

#### Quotient structure: example

- A state of $\mathcal{K}/\sigma$ is an equivalence class of states of $\mathcal{K}$ for $\sigma$.
- 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

- $\mathcal{K}_1 \times \mathcal{K}_2$ 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!
Existential and universal predecessors and successors

Given a relation \( \mathcal{R} \subseteq S_1 \times S_2 \), we define

\[
\begin{align*}
\text{pre}_\exists^\mathcal{R}(Q_2) & := \{ s_1 \in S_1 \mid \exists s_2. (s_1, s_2) \in \mathcal{R} \land s_2 \in Q_2 \} \\
\text{pre}_\forall^\mathcal{R}(Q_2) & := \{ s_1 \in S_1 \mid \forall s_2. (s_1, s_2) \in \mathcal{R} \rightarrow s_2 \in Q_2 \} \\
\text{suc}_\exists^\mathcal{R}(Q_1) & := \{ s_2 \in S_2 \mid \exists s_1. (s_1, s_2) \in \mathcal{R} \land s_1 \in Q_1 \} \\
\text{suc}_\forall^\mathcal{R}(Q_1) & := \{ s_2 \in S_2 \mid \forall s_1. (s_1, s_2) \in \mathcal{R} \rightarrow s_1 \in Q_1 \}
\end{align*}
\]

- \( \text{pre}_\exists^\mathcal{R}(Q_2) \) := the set of states that have a successor in \( Q_2 \)
- \( \text{pre}_\forall^\mathcal{R}(Q_2) \) := the set of states that have no successor in \( S \setminus Q_2 \)
Important properties of predecessors and successors

Duality laws

- \( \text{pre}^R_\equiv (Q_2) := S_1 \setminus \text{pre}^R_\equiv (S_2 \setminus Q_2) \)
- \( \text{pre}^R_\forall (Q_2) := S_1 \setminus \text{pre}^R_\forall (S_2 \setminus Q_2) \)
- \( \text{suc}^R_\forall (Q_1) := S_2 \setminus \text{suc}^R_\forall (S_1 \setminus Q_1) \)
- \( \text{suc}^R_\equiv (Q_1) := S_2 \setminus \text{suc}^R_\equiv (S_1 \setminus Q_1) \)

Monotonicity laws

All these functions are monotonic: e.g.

\[ Q_2 \subseteq Q'_2 \Rightarrow \text{pre}^R_\equiv (Q_2) \subseteq \text{pre}^R_\equiv (Q'_2) \]
Big picture of Verification, Model Checking and Testing

Given:
$M$: the (model) of the system developed
$S$: the (model) of the (correct) system to provide
$\Phi$: a specification of a required property
$T$: a set of correct behaviors (i.e. $[T] \subseteq [S]$)

- Verification checks that $[M] \approx [S]$
- Model checking checks that $[M] \subseteq [\Phi]$
- Testing checks that $[T] \subseteq [M]$

- Generally we do not have $S$
- 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

1. Linear time versus branching time
2. Principle of model checking
3. Linear Temporal Logic
4. Computation Tree Logic
Model Checking: The Basic Algorithm

A Mutual Exclusion Algorithm

- problem setting: find an algorithm such that
  - a group of (two) concurrent processes share a common resource
  - no more than one process has access at the same time
  - access to the resource is modeled by a critical section

- a first simplistic example:
  assert two processes $P_0, P_1$ given as

1. `# non-critical section`
2. `while (other process critical) :
  3. wait()
4. `# critical section`
5. `# return to non-critical`
Mutex: Operational Semantics

- focus mainly “models” that are Kripke structures $\mathcal{M} = \langle S, R, L \rangle$
  (set of state $S$, transitions $R \subseteq S \times S$, labeling $L : S \rightarrow 2^{AP}$, $AP$ is finite set of atomic predicates, no default initial states)

- model mutex algorithm
  - 2 processes $P_0$ and $P_1$ as before
  - generate $\mathcal{M} = \langle S, R, L \rangle$ by product construction
  - write (global) states as $s_0s_1 \in S$, i.e., $P_0$ in $s_0$ and $P_1$ in $s_1$

Mutex: Specifying Properties

Safety: The protocol allows only one process to be in its critical section at any time.
Mutex: Specifying Properties

Safety: The protocol allows only one process to be in its critical section at any time.

Liveness: Whenever any process wants to enter its critical section, it will eventually be permitted to do so.

Non-Blocking: A process can always request to enter its critical section.
Mutex: Specifying Properties

Safety: The protocol allows only one process to be in its critical section at any time.

Liveness: Whenever any process wants to enter its critical section, it will eventually be permitted to do so.

Non-Blocking: A process can always request to enter its critical section.

No Strict Sequencing: Processes need not enter their critical section in a strict sequence.

Mutex: Simplified Properties

- can simplify our properties as $P_0$ and $P_1$ are “identical”:

Safety: The protocol allows only process to be in its critical section at any time.

Liveness: Whenever $P_0$ wants to enter its critical section, it will eventually be permitted to do so.

Non-Blocking: $P_0$ can always request to enter its critical section.

No Strict Sequencing: $P_0$ needs not enter its critical section in a strict sequence with $P_1$. 
Temporal Logics

How to formalize these requirements?

... such that we have a rigorous semantics?
... such that we can verify that they hold?
... such that a tool can help us checking it?

♫ we need to take a look at temporal logics...

Plan

1. Linear time versus branching time
2. Principle of model checking
3. Linear Temporal Logic
4. Computation Tree Logic
Example of linear vs branching semantics

Linear time versus branching time
Principle of model checking
Linear Temporal Logic
Computation Tree Logic

Given the Kripke structure

\[
[S] = \{ \text{traces of } S \} \]
Branching time

Computation tree semantics

\[ [S] = \text{tree given by the unfolding of } S \]

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
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

1. Linear time versus branching time
2. Principle of model checking
3. Linear Temporal Logic
4. Computation Tree Logic

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

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)
  $\Rightarrow [S] \subseteq [\phi]$
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.
Given a set of propositions $\mathcal{P}$, a formula in Linear Temporal Logic (LTL) is defined using the following grammar:

$$\phi ::= T | \perp | p | \neg \phi | \phi \lor \phi | \phi \land \phi | \Diamond \phi | \phi U \phi | \phi \bar{U} \phi$$

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

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

Derived operators

- $\Diamond \equiv T \ U \phi$ (finally).
- $\Box \equiv \neg \Diamond \equiv \perp \bar{U} \phi$ (globally).
Semantics of LTL

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

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

\[
\begin{align*}
\sigma \models T & \iff p \in \mathcal{L}(s_0) \\
\sigma \not\models \bot & \\
\sigma \models p & \iff p \in \mathcal{L}(s_0) \\
\sigma \models \neg \phi & \iff \sigma \not\models \phi \\
\sigma \models \phi_1 \lor \phi_2 & \iff \sigma \models \phi_1 \lor \sigma \models \phi_2 \\
\sigma \models \phi_1 \land \phi_2 & \iff \sigma \models \phi_1 \land \sigma \models \phi_2 \\
\sigma \models \phi & \iff \phi_1 \models \phi \\
\sigma \models \phi_1 \phi_2 & \iff \exists i. \sigma_i \models \phi_2 \land \forall 0 \leq j < i. \sigma_j \models \phi_1 \\
\sigma \models \phi_1 \phi_2 & \iff \forall i \geq 0. \sigma_i \not\models \phi_2 \rightarrow \exists 0 \leq j < i. \sigma_j \models \phi_1
\end{align*}
\]

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 :

\[
\phi ::= T \mid \bot \mid p \mid \neg p \mid \phi \lor \phi \mid \phi \land \phi \mid \phi \mid \phi \cup \phi \mid \phi \phi
\]

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

**Translation extended LTL to restricted LTL**

- $\neg (\phi_1 \cup \phi_2) \equiv (\neg \phi_1) \phi (\neg \phi_2)$
- $\neg (\phi_1 \phi_2) \equiv (\neg \phi_1) \phi (\neg \phi_2)$
- $\neg \phi_1 \equiv \phi \phi_1$
Linear time versus branching time
Principle of model checking
Linear Temporal Logic
Computation Tree Logic

Plan

1. Linear time versus branching time
2. Principle of model checking
3. Linear Temporal Logic
4. Computation Tree Logic

LTL model checking

Principle

- \( S \models \phi \equiv [S] \subseteq [\phi] \equiv [S] \cap [\neg \phi] = \emptyset \)
- 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).
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**

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

$$\phi ::= T \mid p \mid \neg \phi \mid \phi \lor \phi \mid \exists \circ \phi \mid \forall \circ \phi \mid \exists \phi \mathbin{U} \phi \mid \forall \phi \mathbin{U} \phi$$

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

- Most of the operators are standard;
- $\exists \circ$ is the exists next operator; $\exists \circ \phi$ is true if there is a path (from the current state) where $\phi$ is true after the first state of the path;
- $U$ is the until operator; $\exists \phi \mathbin{U} \psi$ is true if there is a path where $\phi$ is true in all the states preceding one state where $\psi$ is true.
- $\forall \circ$ and $\forall U$ are similar but for all paths.
Semantics of CTL (on tree and uses traces $\sigma = s_0s_1s_2 \ldots$)

$\begin{align*}
    s_0 &\models T \\
    s_0 &\models p \quad \text{iff} \quad p \in \mathcal{L}(s_0) \\
    s_0 &\models \neg \phi \quad \text{iff} \quad s_0 \not\models \phi \\
    s_0 &\models \phi_1 \lor \phi_2 \quad \text{iff} \quad s_0 \models \phi_1 \lor s_0 \models \phi_2 \\
    s_0 &\models \exists \bigcirc \phi \quad \text{iff} \quad \exists s_1.s_0 \xrightarrow{\kappa} s_1 \land s_1 \models \phi \\
    s_0 &\models \forall \bigcirc \phi \quad \text{iff} \quad \forall s_1.s_0 \xrightarrow{\kappa} s_1 \rightarrow s_1 \models \phi \\
    s_0 &\models \exists \phi_1 \bigtriangledown \phi_2 \quad \text{iff} \quad \exists \sigma \in \text{Path}(s_0).\exists i.\sigma' \models \phi_2 \land \forall 0 \leq j < i.\sigma' \models \phi_1 \\
    s_0 &\models \forall \phi_1 \bigtriangledown \phi_2 \quad \text{iff} \quad \forall \sigma \in \text{Path}(s_0).\exists i.\sigma' \models \phi_2 \land \forall 0 \leq j < i.\sigma' \models \phi_1
\end{align*}$

CTL syntax (cont’d)

**Derived operators**

- $\exists \bigtriangledown \phi \equiv \exists T \bigcirc \phi$ (exists finally).
- $\forall \bigtriangledown \phi \equiv \forall T \bigcirc \phi$ (for all finally).
- $\exists \square \phi \equiv \neg \exists \bigtriangledown \neg \phi$ (exists globally).
- $\forall \square \phi \equiv \neg \forall \bigtriangledown \neg \phi$ (for all globally).

**CTL without universal quantifiers (is sometimes useful)**

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

- $\forall \bigcirc \phi \equiv \neg \exists \bigcirc \neg \phi$
- $\forall \phi_1 \bigtriangledown \phi_2 \equiv \neg \exists (\neg \phi_2 \bigtriangledown (\neg \phi_1 \land \neg \phi_2)) \land \neg \exists (\square \neg \phi_2)$
- $\forall \square \phi \equiv \neg \exists \bigtriangledown \neg \phi$
- $\forall \bigtriangledown \phi \equiv \neg \exists \square \neg \phi$
Safety: The protocol allows only one process to be in its critical section at any time.

\[
\varphi_{\text{safety}} \equiv AG\neg(c_0 \land c_1)
\]
Safety: \( \varphi_{\text{safety}} \equiv AG \neg(c_0 \land c_1) \)

Liveness: Whenever \( P_0 \) wants to enter its critical section, it will eventually be permitted to do so.

Safety: \( \varphi_{\text{safety}} \equiv AG \neg(c_0 \land c_1) \)

Liveness: \( \varphi_{\text{liveness}} \equiv AG(t_0 \rightarrow AF c_0) \)
Safety: \[ \varphi_{\text{safety}} \equiv AG \neg (c_0 \land c_1) \]

Liveness: \[ \varphi_{\text{liveness}} \equiv AG (t_0 \rightarrow AF c_0) \]

Non-Blocking: \[ \varphi_{\text{nblock}} \equiv AG (n_0 \rightarrow EX t_0) \]

\[ P_0 \text{ can always request to enter its critical section.} \]
CTL: Back to Mutex Example

Safety: \( \varphi_{\text{safety}} \equiv AG \neg (c_0 \land c_1) \)

Liveness: \( \varphi_{\text{liveness}} \equiv AG (t_0 \rightarrow AF c_0) \)

Non-Blocking:
\( \varphi_{\text{nblock}} \equiv AG (n_0 \rightarrow EX t_0) \)

No Strict Sequencing:
\( P_0 \) needs not enter their critical section in a strict sequence with \( P_1 \).
CTL model checking

Principle

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

Chapter 4: $\omega$-automata

1. Motivation
2. Büchi automata
3. Properties of Büchi automata
4. From LTL to Büchi automata
Motivation

Büchi automata

Properties of Büchi automata

From Ltl to Büchi automata

Need to extend Finite automata

Finite and infinite words

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

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
Büchi automaton

\[ \mathcal{A} = (\Sigma, Q, I, R, F) \]

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

Deterministic Büchi automaton

- \[ I = \{ q_1 \} \]
- \[ |\{ q' \mid \exists q \in Q, \sigma \in \Sigma. (q, \sigma, q') \in R \}| = 1 \]
Buchi acceptance condition

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

\( w \) accepted by \( \mathcal{A} \)
An infinite word \( w \) is accepted by the automaton \( \mathcal{A} \) if there exists an infinite path \( \rho : \rho : \mathbb{N} \mapsto Q \) such that
- \( \rho(0) \in I \) (the path starts at an initial state)
- \( \forall i \in \mathbb{N}. (\rho(i), w_i, \rho(i + 1)) \in \mathcal{R} \)
- \( \inf(\rho) \cap F \neq \emptyset \)
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

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

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

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

\( \mathcal{L}(\mathcal{A}_4) = (a + b)^*a(b(a + b)^*a)^\omega = ((a + b)^*ab)^\omega \)
Other types of acceptance conditions

- **Generalized Büchi**: $\mathcal{F} \subseteq 2^Q$, i.e. $\mathcal{F} = \{F_1, \ldots, F_m\}$
  For each $F_i$, $\inf(\rho) \cap F_i \neq \emptyset$

- **Rabin**: $\mathcal{F} \subseteq 2^Q \times 2^Q$, i.e. $\mathcal{F} = \{(G_1, B_1), \ldots, (G_m, B_m)\}$
  For some pair $(G_i, B_i) \in \mathcal{F}$, $\inf(\rho) \cap G_i \neq \emptyset \land \inf(\rho) \cap B_i = \emptyset$

- **Streett**: $\mathcal{F} \subseteq 2^Q \times 2^Q$, i.e. $\mathcal{F} = \{(G_1, B_1), \ldots, (G_m, B_m)\}$
  For all pairs $(G_i, B_i) \in \mathcal{F}$, $\inf(\rho) \cap G_i = \emptyset \lor \inf(\rho) \cap B_i \neq \emptyset$

- **Parity**: with $Q = \{0, 1, 2, \ldots, k\}$ for some natural $k$, $A$ accepts $\rho$ iff the smallest number in $\text{Inf}(\rho)$ is even (where $\text{Inf}(\rho)$ is the set of states that occur infinitely often).

- **Muller**: $\mathcal{F} \subseteq Q$, $|\inf(\rho) \cap \mathcal{F}| = 1$

---

**ω-regular languages**

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

---

**Plan**

- **Motivation**
- **Büchi automata**
- **Properties of Büchi automata**
- **From Ltl to Büchi automata**
Properties of Büchi automata

Büchi automata are closed under union
- Given $A_1$ and $A_2$ with disjoint states set $Q_1$ and $Q_2$
- It is easy to define $A$ with
  - union of states,
  - union of initial states
  - union of accepting states
  - union of transition relation
- $\mathcal{L}(A) = \mathcal{L}(A_1) \cup \mathcal{L}(A_2)$

Büchi automata are closed under intersection
- Given the Büchi automata $A_1 = (\Sigma, Q_1, I_1, R_1, F_1)$ and $A = (\Sigma, Q_2, I_2, R_2, F_2)$
- We can define $A = (\Sigma, Q, I, R, F)$ with
  - $Q = Q_1 \times Q_2 \times \{1, 2\}$,
  - $I = I_1 \times I_2 \times \{1\}$,
  - $F = \{F_1 \times Q_2 \times \{1\}$
  - $\forall s, s' \in Q_1, t, t' \in Q_2, a \in \Sigma, i, j \in \{1, 2\}$:
    - $((s, t, i), a, (s', t', j)) \in R$ $\iff$ $(s, a, s') \in R_1 \land (t, a, t') \in R_2$, and
    - $i = 1, s \in F_1$, and $j = 2$ or
    - $i = 2, t \in F_2$, and $j = 1$ or
    - neither 1 or 2 above applies and $j = i$
- $\mathcal{L}(A) = \mathcal{L}(A_1) \cap \mathcal{L}(A_2)$
Properties of Büchi automata

Büchi automata are closed under complementation
- difficult! (not seen here)

Nonemptyness 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

Given $\mathcal{A} = (\Sigma, Q, I, R, F)$ where $F = \{F_1, \ldots, F_k\}$
$A' = (\Sigma, Q', I', R', F')$ where
- $Q' = Q \times \{1, \ldots, k\}$
- $I' = I \times \{1\}$
- $R'$ is defined by $((s, j), a, (t, i)) \in R'$ if $(s, a, t) \in R$ and
  - $i = j$ if $s \not\in F_i$
  - $i = (j \mod k) + 1$ if $s \in F_i$
- $F' = F_1 \times \{1\}$

$A \equiv A'$
From generalized Büchi to Büchi, example

\[ \mathcal{F} = \{ \{s_0\}, \{s_1\} \} \]

\[ \mathcal{F} = \{(s_0, 1)\} \]
Given a (restricted) LTL formula $\phi$. We want to build a Büchi automaton $A_\phi$ with

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

**Construction of the Büchi automaton of an LTL formula $\phi$**

**Principle on the construction of $A_\phi$**

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

4 steps:
- Construction of the **local automaton** for $\phi$
- Construction of the **eventualities automaton** for $\phi$
- Composition of both automaton to build a Generalized Büchi automaton
- Transformation for the result into a simple Büchi automaton

Closure of $\phi$

- $\phi \in cl(\phi)$
- $\neg \phi \in cl(\phi)$
- $\phi_1 \land \phi_2 \in cl(\phi) \Rightarrow \phi_1, \phi_2 \in cl(\phi)$
- $\phi_1 \lor \phi_2 \in cl(\phi) \Rightarrow \phi_1, \phi_2 \in cl(\phi)$
- $\bigcirc \phi \in cl(\phi) \Rightarrow \phi \in cl(\phi)$
- $\phi_1 \cup \phi_2 \in cl(\phi) \Rightarrow \phi_1, \phi_2 \in cl(\phi)$
- $\phi_1 \bar{U} \phi_2 \in cl(\phi) \Rightarrow \phi_1, \phi_2 \in cl(\phi)$
Construction of the local automaton for $\mathcal{A}_\phi$

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

The set of states are all “compatible” subset of $\text{cl}(\phi)$:

- $\phi_1 \land \phi_2 \in s \Rightarrow \phi_1 \in s \land \phi_2 \in s$
- $\phi \in s \iff \neg \phi \notin s$
- $\phi_1 \lor \phi_2 \in s \Rightarrow \phi_1 \in s \lor \phi_2 \in s$
- $\phi_1 \cup \phi_2 \in s \Rightarrow \phi_1 \in s \lor \phi_2 \in s$
- $\phi_1 \tilde{\cup} \phi_2 \in s \Rightarrow \phi_2 \in s$

**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 $\phi_1 \cup \phi_2 \in s \land \phi_2 \notin s \Rightarrow \phi_1 \cup \phi_2 \in t$
- If $\phi_1 \tilde{\cup} \phi_2 \in s \land \phi_1 \notin s \Rightarrow \phi_1 \tilde{\cup} \phi_2 \in t$

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

The local automaton for $p \cup q$

![Diagram](image-url)
Construction of local automaton for $A_\phi$

The local automaton for $p \lor q$

![Diagram of local automaton for $p \lor q$]

local automaton for $p \lor q$

Construction of eventualities automaton for $A_\phi$

Goal: check that all the $\phi_1 \lor \phi_2$ are “finalized”, i.e. for each $\phi_1 \lor \phi_2$ find a state where $\phi_2$ is true.

**eventualities of $\phi$: $ev(\phi)$**
- subset of $cl(\phi)$ of the form $\phi_1 \lor \phi_2$

**Eventualities automaton for $\phi$**

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

- $s = \emptyset \Rightarrow \forall \phi_1 \lor \phi_2 \in ev(\phi): \phi_1 \lor \phi_2 \in t \iff \phi_2 \notin A$
- $s \neq \emptyset \Rightarrow \forall \phi_1 \lor \phi_2 \in s: \phi_1 \lor \phi_2 \in t \iff \phi_2 \notin A$
Motivation
Büchi automata
Properties of Büchi automata
From LTL to Büchi automata

Construction of eventualities automaton for $A_\phi$

The eventualities automaton for $p U q$

![Eventualities automaton for $p U q$]

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

![Construction of $A_\phi$ for $p U q$]
Chapter 5: $\mu$-calculus

1. Elements of Lattice theory

2. $\mu$-calculus
Order and partially ordered sets (posets)

Order
- **Order**: binary relation over \( D \) with the properties of
  - Reflexivity
  - Antisymmetry
  - Transitivity
- **Total order**: order with \( \forall x, y \in D. x \sqsubseteq y \lor y \sqsubseteq x \)

Partial order set (or poset)

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

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

Elements of Lattice theory

(least) upper bound and (greatest) lower bound

Given a poset \((D, \sqsubseteq)\)

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

- $\bigcup(\bigcup_{i \in I} A_i) = \bigcup(\bigcup_{i \in I} A_i)$
- $\bigcap(\bigcup_{i \in I} A_i) = \bigcap(\bigcup_{i \in I} A_i)$
- $A \subseteq B$ implies $\bigcup A \subseteq \bigcup B$
- $A \subseteq B$ implies $\bigcap B \subseteq \bigcap A$

(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 $\{x, y\}$ and $\bigcap \{x, y\}$
  - $(\mathcal{D}, \sqsubseteq)$ is a complete lattice if for all non empty $M \subseteq \mathcal{D}$ have $\bigcup M$ and $\bigcap M$
- In a lattice, for every finite set $M_{\text{fin}}$
  - $\bigcup \{e\} \cup M_{\text{fin}} = \bigcup \{e, \bigcup M_{\text{fin}}\}$
  - $\bigcap \{e\} \cup M_{\text{fin}} = \bigcap \{e, \bigcap M_{\text{fin}}\}$
- In a lattice, $\bigcup M_{\text{fin}}$ and $\bigcap M_{\text{fin}}$ exist for finite $M_{\text{fin}}$
- In a complete lattice, we define $\bot := \bigcap \mathcal{D}$ and $\top := \bigcup \mathcal{D}$
Example of lattice and complete lattice

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

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

Elements of Lattice theory

\(\mu\)-calculus

Algebraic properties of Lattices

Given \(x, y\)

- **Commutativity**: \(x \cap y = y \cap x\) \quad (x \cup y = y \cup x)
- **Associativity**: \(x \cap (y \cap z) = (x \cap y) \cap z\) \quad (x \cup (y \cup z) = (x \cup y) \cup z)
- **Absorption**: \(x \cap (x \cup y) = x\) \quad (x \cup (x \cap y) = x)
- **Idempotency**: \(x \cap x = x\) \quad (x \cup x = x)
Important lattice: The set of subsets of a set $S$

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

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

Monotonic and Continuous Functions and fixpoint

- Given complete lattices $(\mathcal{D}, \sqsubseteq)$ and $(\mathcal{E}, \sqsubseteq)$
- Given a function $f : \mathcal{D} \to \mathcal{E}$
- $f$ is monotonic if $x \sqsubseteq y \implies f(x) \sqsubseteq f(y)$
- $f$ 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 \( D \) a complete lattice and continuous \( f : D \to D \)
- 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 \( \nu x. f(x) \)

\[ \begin{align*}
\mu x. f(x) &= \bigcap \{ x \in D \mid f(x) \sqsubseteq x \} \\
\nu x. f(x) &= \bigcup \{ x \in D \mid f(x) \sqsupseteq x \}
\end{align*} \]

Computation of fixpoints with finite complete lattice

Given a finite lattice \( (D, \sqsubseteq) \) 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 \( M \)
What is the \( \mu \)-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 \( \mu \)-calculus (e.g. LTL, CTL, CTL*).

Syntax of the \( \mu \)-calculus

Note: the following definition is a possible \( \mu \)-calculus; other operators could be defined.

Set of \( \mu \)-calculus formulae \( \mathcal{L}_\mu \)

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

\[ \top | \bot | x | \neg \phi | \phi \land \psi | \phi \lor \psi | \langle \rangle \phi | [\[] \phi | \mu x. \phi | \nu x. \phi \]
Semantics of the \(\mu\)-calculus

Given

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

\([\phi]_{\mathcal{K}}\) gives the set of states which satisfies \(\phi\)

- \([\top]_{\mathcal{K}} := S\)
- \([\bot]_{\mathcal{K}} := \emptyset\)
- \([x]_{\mathcal{K}} := \{s \in S \mid x \in L(s)\}\)
- \([-\phi]_{\mathcal{K}} := S \setminus [\phi]_{\mathcal{K}}\)
- \([\phi \land \psi]_{\mathcal{K}} := [\phi]_{\mathcal{K}} \cap [\psi]_{\mathcal{K}}\)
- \([\phi \lor \psi]_{\mathcal{K}} := [\phi]_{\mathcal{K}} \cup [\psi]_{\mathcal{K}}\)
- \([\langle \rangle \phi]_{\mathcal{K}} := \text{pre}_R^\mathcal{K}(\phi)\) (set of states which have a successor in \([\phi]_{\mathcal{K}}\))
- \([\downarrow \uparrow \phi]_{\mathcal{K}} := \text{pre}^\mathcal{K}_R(\phi)\) (set of states which have all successors in \([\phi]_{\mathcal{K}}\))

Modified structure \(\mathcal{K}_x^Q\)

Given the Kripke structure \(\mathcal{K} = (\mathcal{I}, S, R, L)\) and a set of states \(Q \subseteq S\), 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

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

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

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 \([\phi]_{\mathcal{K},\mathcal{Q}}\)**

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

<table>
<thead>
<tr>
<th>(Q)</th>
<th>([\phi]_{\mathcal{K},\mathcal{Q}})</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\emptyset)</td>
<td>({s_3})</td>
</tr>
<tr>
<td>({s_2, s_3})</td>
<td>({s_1, s_2, s_3, s_4})</td>
</tr>
<tr>
<td>({s_0, s_1, s_2, s_3, s_4})</td>
<td>({s_0, s_1, s_2, s_3, s_4})</td>
</tr>
</tbody>
</table>

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

**Example of \([\phi]_{\mathcal{K},\mathcal{Q}}\)**

With \(\mathcal{K}\) and \(\phi = [\langle \rangle (x \land q)\)

<table>
<thead>
<tr>
<th>(Q)</th>
<th>([\phi]_{\mathcal{K},\mathcal{Q}})</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\emptyset)</td>
<td>(\emptyset)</td>
</tr>
<tr>
<td>({s_0, s_1, s_2, s_3})</td>
<td>({s_0, s_1, s_2, s_3})</td>
</tr>
<tr>
<td>({s_0, s_1})</td>
<td>({s_0, s_1})</td>
</tr>
<tr>
<td>({s_0})</td>
<td>(\emptyset)</td>
</tr>
</tbody>
</table>

For \(Q = \emptyset\),
\(Q = [\phi]_{\mathcal{K},\mathcal{Q}}\) (\(Q\) is a fixpoint for \([\phi]_{\mathcal{K},\mathcal{Q}}\))
Examples:

Example of $[\phi]_{K-Q}$

With $K$ and $\phi = \neg x$

<table>
<thead>
<tr>
<th>$Q$</th>
<th>$[\phi]_{K-Q}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\emptyset$</td>
<td>$S$</td>
</tr>
<tr>
<td>$S$</td>
<td>$\emptyset$</td>
</tr>
<tr>
<td>${s_0}$</td>
<td>${s_1}$</td>
</tr>
<tr>
<td>${s_1}$</td>
<td>${s_0}$</td>
</tr>
</tbody>
</table>

No fixpoint here!

Semantics of the $\mu$-calculus

Semantics II

Given

- $[\top]_K := S$
- $[\bot]_K := \emptyset$
- $[x]_K := \{s \in S \mid x \in \mathcal{L}(s)\}$
- $[\neg \phi]_K := S \setminus [\phi]_K$
- $[\phi \land \psi]_K := [\phi]_K \cap [\psi]_K$
- $[\phi \lor \psi]_K := [\phi]_K \cup [\psi]_K$
- $[\langle \phi \rangle]_K := \text{pre}^R_S([\phi]_K)$ (set of states which have a successor in $[\phi]$)
- $[\{\} \phi]_K := \text{pre}^R_S([\phi]_K)$ (set of states which have all successors in $[\phi]$)
- $[\mu x. \phi]_K$ is the least fixpoint of $f(Q) := [\phi]_{K-Q}$
- $[\nu x. \phi]_K$ is the greatest fixpoint of $f(Q) := [\phi]_{K-Q}$
Existence of fixpoints

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

Example of $\mu$-calculus formulae

Properties specified by $\mu$-calculus formulae

- Invariance ($\phi$ always true) : $\nu x. (\phi \land [\top] x)$
- Reachability ($\phi$ reachable) : $\mu x. (\phi \lor (\top) x)$
- Persistence ($\phi$ is reachable and then remains always true) : $\mu y. [\nu x. (\phi \land [\top] x) \lor (\top) y]$
**Principle**
- Compute the set of states in $\mathcal{K}$, which satisfy subformulae $\phi_i$ of $\phi$
- $\mathcal{K} \models \phi \iff I \subseteq [\phi]_\mathcal{K}$
- see chapter 6.
LTL model checking

- LTL model checking
- Ctl model checking
- μ-calculus 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:

$$
\phi ::= T | \bot | p | \neg \phi | \phi \lor \psi | \phi \land \psi | \Box \phi | \phi U \psi | \phi \bar{U} \psi
$$

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

Principle of LTL model checking

- $S \models \phi \equiv [S] \subseteq [\phi] \equiv [S] \cap [\neg \phi] = \emptyset$
- 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 $\mathcal{L}(S \times B_{\neg \phi}) = \emptyset$
LTL model checking

Complexity of the LTL model checking

- The size of the Büchi automaton for $\neg \phi$ is (in the worst case), in $O(2^{|\phi|})$
- The size of $S \times B_{\neg \phi}$ is in $O(|S| |B_{\neg \phi}|)$
- Checking if $L(S \times B_{\neg \phi}) = \emptyset$ is linear in the size of $S \times B_{\neg \phi}$
- The resulting complexity is in $O(|S| 2^{|\phi|})$ (linear in the size of the system, exponential in the size of the formula)
From previous chapter

**CTL syntax**

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

$$\phi ::= T \mid p \mid \neg \phi \mid \phi \lor \phi \mid \exists \phi \mid \forall \phi \mid \exists \phi U \phi \mid \forall \phi U \phi$$

where $p \in P$.

**Principle of CTL model checking**

- For all state $s$ in $K$, decorate $s$ with all the subformulae $\phi_i$ of $\phi$ such that $s |= \phi_i$

**CTL without universal quantifiers**

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

- $\forall \bigcirc \phi \equiv \neg \exists \bigcirc \neg \phi$
- $\forall \phi_1 U \phi_2 \equiv \neg \exists (\neg \phi_2 U (\neg \phi_1 \land \neg \phi_2)) \land \neg \exists \bigcirc \neg \phi_2$
- $\forall \square \phi \equiv \neg \exists \bigcirc \neg \phi$
- $\forall \diamond \phi \equiv \neg \exists \bigcirc \neg \phi$
CTL model checking

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

Structure of the algorithm

- Inductive:
  - basis:
    - $T$ 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 $\phi \notin \mathcal{L}(s)$
    - $\phi_1 \lor \phi_2 \in \mathcal{L}(s)$ iff $\phi_1 \in \mathcal{L}(s) \lor \phi_2 \in \mathcal{L}(s)$
    - $\exists \circ \phi \in \mathcal{L}(s)$ iff $\exists t \in \text{suc}_R(s).\phi \in \mathcal{L}(t)$
    - $\forall \circ \phi \in \mathcal{L}(s)$ iff $\forall t \in \text{suc}_R(s).\phi \in \mathcal{L}(t)$
    - $\exists \phi_1 \lor \phi_2$: see algorithm below
    - $\forall \phi_1 \lor \phi_2$: see algorithm below

Algorithm for $s \models \forall \phi_1 \lor \phi_2$

 Require: $\forall t \in S.\neg \text{marked}(t) \land \forall i \in \{1, 2\}.(\phi_i \in \mathcal{L}(t) \iff t \models \phi_i)$

 Ensure: return true $\land (\forall \phi_1 \lor \phi_2 \in \mathcal{L}(s) \iff s \models \forall \phi_1 \lor \phi_2)$

 if $(\forall \phi_1 \lor \phi_2) \in \mathcal{L}(s)$ then
   return true
 else if $\neg \text{marked}(s)$ then
   if $\phi_2 \in \mathcal{L}(s)$ then
     $\mathcal{L}(s) \leftarrow (\forall \phi_1 \lor \phi_2)$
     return true
   else if $\phi_1 \notin \mathcal{L}(s)$ then
     return false
   else
     $\text{marked}(s) \leftarrow \text{true}$
     if $\forall t \in \text{suc}_R(s).t \models \forall \phi_1 \lor \phi_2$ then
       $\mathcal{L}(s) \leftarrow (\forall \phi_1 \lor \phi_2)$
       return true
     else
       return false
     end if
   end if
 else
   return false
end if
Algorithm for $s \models \exists \phi_1 U \phi_2$

**Require:** $\forall t \in S. \neg \text{marked}(t) \land \forall i \in \{1, 2\}. (\phi_i \in \mathcal{L}(t) \iff t \models \phi_i)$

**Ensure:** return true $\land (\exists \phi_1 U \phi_2 \in \mathcal{L}(s) \iff s \models \exists \phi_1 U \phi_2)$

if $(\forall \phi_1 U \phi_2) \in \mathcal{L}(s)$ then
    return true
else if $\neg \text{marked}(s)$ then
    if $\phi_2 \in \mathcal{L}(s)$ then
        $\mathcal{L}(s) \leftarrow (\exists \phi_1 U \phi_2)$
        return true
    else if $\phi_1 \notin \mathcal{L}(s)$ then
        return false
    else
        $\text{marked}(s) \leftarrow \text{true}$
        if $\exists t \in \text{suc}_R(s). t \models \exists \phi_1 U \phi_2$ then
            $\mathcal{L}(s) \leftarrow (\exists \phi_1 U \phi_2)$
            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 $\mu$-calculus translation (see below)
Plan

1. Ltl model checking

2. Ctl model checking

3. \(\mu\)-calculus model checking

From previous chapter

\(\mu\)-calculus syntax

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

\[ T | \bot | p | x | \neg \phi | \phi \land \psi | \phi \lor \psi | \langle \rangle \phi | [\ ] \phi | \mu x. \psi | \nu x. \psi \]

Principle of the \(\mu\)-calculus model checking

- Compute the set of states in \(\mathcal{K}\), which satisfy subformulae \(\phi_i\) of \(\phi\)
- \(\mathcal{K} \models \phi \iff \mathcal{I} \subseteq [\phi]_\mathcal{K}\)
Algorithm which computes $States_\mu(\phi)$

Case $\phi \equiv$

- $\top$ : return $S$
- $\bot$ : return $\emptyset$
- $x : $ return $\{ s \in S \mid x \in \mathcal{L}(s) \}$
- $\neg \phi_1$ : return $S \setminus States_\mu(\phi_1)$
- $\phi_1 \land \phi_2$ : return $States_\mu(\phi_1) \cap States_\mu(\phi_2)$
- $\phi_1 \lor \phi_2$ : return $States_\mu(\phi_1) \cup States_\mu(\phi_2)$
- $\langle \rangle \phi_1$ : return $\text{pre}_\mathcal{R} \left( States_\mu(\phi_1) \right)$
- $[\ ] \phi_1$ : return $\text{pre}_\mathcal{L} \left( States_\mu(\phi_1) \right)$

$\mu x. \psi : Q_1 := \{ \}; \quad \nu x. \psi : Q_1 := S$

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

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

LTL model checking

CTL model checking through a translation into the $\mu$-calculus

CTL to $\mu$-calculus (Clarke and Emerson 1981)

- $\exists \bigcirc \phi = \langle \rangle \phi$
- $\forall \bigcirc \phi = [\ ] \phi$
- $\exists \phi_1 \lor \phi_2 = \phi_2 \lor (\phi_1 \land \langle \exists \phi_1 \lor \phi_2 \rangle) = \mu Z. [\phi_2 \lor (\phi_1 \land \langle \rangle Z)]$
- $\forall \phi_1 \lor \phi_2 = \phi_2 \lor (\phi_1 \land [\forall \phi_1 \lor \phi_2]) = \mu Z. [\phi_2 \lor (\phi_1 \land [\ ] Z)]$
- $\exists \langle \bigcirc \phi_1 \rangle = \phi_1 \lor (\exists \bigcirc \phi_1) = \mu Z. [\phi_1 \lor (\langle \rangle Z)]$
- $\forall \langle \bigcirc \phi_1 \rangle = \phi_1 \lor (\forall \bigcirc \phi_1) = \mu Z. [\phi_1 \lor (\ [\ ] Z)]$
- $\exists \bigcirc \phi_1 = \phi_1 \land (\exists \bigcirc \phi_1) = \nu Z. [\phi_1 \land (\langle \rangle Z)]$
- $\forall \bigcirc \phi_1 = \phi_1 \land (\forall \bigcirc \phi_1) = \nu Z. [\phi_1 \land (\ [\ ] Z)]$
CTL model checking through a translation into the $\mu$-calculus

### CTL to $\mu$-calculus with no use of the $[\cdot]$ operator

Through the replacement of every universal quantifiers:

- $\forall \bigotimes \phi \equiv \neg \exists \bigotimes \neg \phi$
- $\forall \phi_1 U \phi_2 \equiv \neg \exists (\neg \phi_2 U (\neg \phi_1 \land \neg \phi_2)) \land \neg \exists \Box \neg \phi_2$
- $\forall \Box \phi \equiv \neg \exists \Diamond \neg \phi$
- $\forall \Diamond \phi \equiv \neg \exists \Box \neg \phi$

### CTL to $\mu$-calculus (Clarke and Emerson 1981)

- $\exists \bigotimes \phi = (\langle \rangle \phi$
- $\forall \bigotimes \phi = [\cdot] \phi$
- $\exists \phi_1 U \phi_2 = \phi_2 \lor (\phi_1 \land (\exists \phi_1 U \phi_2)) = \mu Z.[\phi_2 \lor (\phi_1 \land (\langle \rangle Z)]$
- $\forall \phi_1 U \phi_2 = \phi_2 \lor (\phi_1 \land [\cdot](\forall \phi_1 U \phi_2)) = \mu Z.[\phi_2 \lor (\phi_1 \land [\cdot]Z)]$
- $\exists \Diamond \phi_1 = \phi_1 \lor (\langle \rangle \exists \Diamond \phi_1) = \mu Z.[\phi_1 \lor (\langle \rangle Z]$
- $\forall \Diamond \phi_1 = \phi_1 \lor (\langle \rangle \forall \Diamond \phi_1) = \mu Z.[\phi_1 \lor (\langle \rangle Z]$
- $\exists \Box \phi_1 = \phi_1 \land (\exists \Box \phi_1) = \nu Z.[\phi_1 \land (\langle \rangle Z]$
- $\forall \Box \phi_1 = \phi_1 \land (\forall \Box \phi_1) = \nu Z.[\phi_1 \land (\langle \rangle Z]$

Chapter 7 : Symbolic and efficient Model Checking

1. Symbolic model checking with BDD
2. Model checking with partial order reduction
3. Model checking with symmetry reduction
4. Bounded model checking
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,
  
  ⇒ 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 \( n = 5 \) : \( v_1 \land \neg v_2 \land v_3 \land v_4 \land \neg v_5 \))
- A set of states \( p \) defined as a predicate.

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

with \( \ell_{ij} = \) either \( v_j \) or \( \neg v_j \)

E.g.
(\( v_1 \land \neg v_2 \land v_3 \land \neg v_4 \land v_5 \)) \lor (\neg v_1 \land v_2 \land \neg v_3 \land v_4 \land v_5 )

- Suppose \( p \) an encoding of this formula for \( p \)
  
  We note : \( p = \lambda(v)p = \lambda(v_1, v_2, v_3, v_4, v_5)p \)
Binary encoding of a transition relation

- A transition $= p \times p'$
- A transition defined by a conjunction of $2n$ literals: e.g.
  $$(v_1 \land \neg v_2 \land v_3 \land \neg v_4 \land v_5) \land (\neg v'_1 \land v'_2 \land \neg v'_3 \land v'_4 \land v'_5)$$
- A transition relation: set of transitions encoded as a predicate.
  $$\bigvee_{i \in \{1 \ldots |R|\}} \bigwedge_{1 \leq j \leq n} \ell_{ij} \land \ell'_{ij}$$
  with $\ell_{ij}$ is either $v_j$ or $\neg v_j$ and $\ell'_{ij}$ is either $v'_j$ or $\neg v'_j$

E.g.
$$(v_1 \land \neg v_2 \land v_3 \land \neg v_4 \land v_5) \land (\neg v'_1 \land v'_2 \land \neg v'_3 \land v'_4 \land v'_5) \lor$$
$$(v_1 \land v_2 \land \neg v_3 \land \neg v_4 \land v_5) \land (v'_1 \land v'_2 \land \neg v'_3 \land \neg v'_4 \land v'_5)$$

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

Computing $\llbracket \langle \phi \rangle \rrbracket$ through its encoding

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

- Given $p$ an encoding of $\llbracket \phi \rrbracket$ (the set of global states which satisfy $\phi$ 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 $\llbracket \langle \phi \rangle \rrbracket$ $= \lambda v. \exists v'(R \land p')$
Example 1: computing the predicate for $[\langle \rangle \neg b]$ in $K$

- $s_1 = \neg b$
- $s_2 = b$
- $R = ((\neg b \land b') \lor (b \land \neg b')) 
\lor (b \land b')) = (b \lor b')$

\[
\langle \rangle \neg b = \exists b' ((b \lor b') \land ((\neg b)[b \leftarrow b']))
\]
\[
= \exists b' ((b \lor b') \land \neg b')
\]
\[
= \exists b' (b \land \neg b')
\]
\[
= (b \land \neg 0) \lor (b \land \neg 1)
\]
\[
= b \text{ (state } s_2)\]

Example 2: computing the predicate for $[\exists \diamond b] = [\mu y. (b \lor \langle \rangle y)]$ in $K$

- $s_1 = \neg b$
- $s_2 = b$
- $R = ((\neg b \land b') \lor (b \land \neg b')) 
\lor (b \land b')) = (b \lor b')$

\[
f(0) = b \lor \langle \rangle 0 = b
\]
\[
f^1(0) = b \lor \langle \rangle b =
\]
\[
b \lor \exists b' ((b \lor b') \land b')
\]
\[
= b \lor (b \lor 1)
\]
\[
= 1 \text{ (all states) }
\]
\[
f^3(0) = b \lor \langle \rangle 1 = 1
\]
CTL model checking revisited

\[ \phi \] : predicate of a CTL formula \( \phi \)

Given \( [s] \) the predicate for \( s \in S \)

\[
\begin{array}{c|c|c}
\top & \forall_{s \in S}[s] & \forall_{s \in S, p \in \mathcal{L}(s)}[s] \\
p & \neg[\phi] & \neg[\phi] \\
\neg \phi & & \\
\phi_1 \lor \phi_2 & [\phi_1] \lor [\phi_2] & [\phi_1] \lor [\phi_2] \\
\exists \Box \phi_1 & \text{EvalEX}([\phi_1]) & \text{EvalEX}([\phi_1]) \\
\exists \phi_1 \lor \phi_2 & \text{EvalEU}([\phi_1], [\phi_2]) & \text{EvalEU}([\phi_1], [\phi_2]) \\
\exists \Box \phi_1 & \text{EvalEF}([\phi_1]) & \text{EvalEF}([\phi_1]) \\
\exists \Box \phi_1 & \text{EvalEG}([\phi_1]) & \text{EvalEG}([\phi_1]) \\
\end{array}
\]

where \( [s] \) is the predicate corresponding to state \( s \).

CTL model checking revisited

\[ \phi \] : predicate of a CTL formula \( \phi \) (cont’d)

\[
\text{EvalEX}(p) := \exists y'(R \land p')
\]

\[
\text{EvalEU}(p, q) :=
\begin{align*}
y &= \emptyset \\
y' &= q \lor (p \land \text{EvalEX}(y)) \\
\text{while}(y \neq y') & \\
y &= y' \\
y' &= q \lor (p \land \text{EvalEX}(y)) \\
\end{align*}
\]

\[
\text{return } y
\]

\[
\text{EvalEF}(p) :=
\begin{align*}
y &= \emptyset \\
y' &= p \lor \text{EvalEX}(y) \\
\text{while}(y \neq y') & \\
y &= y' \\
y' &= p \lor \text{EvalEX}(y) \\
\end{align*}
\]

\[
\text{return } y
\]

\[
\text{EvalEG}(p) :=
\begin{align*}
y &= \top \\
y' &= p \land \text{EvalEX}(y) \\
\text{while}(y \neq y') & \\
y &= y' \\
y' &= p \land \text{EvalEX}(y) \\
\end{align*}
\]

\[
\text{return } y
\]
**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 \( x \):
  \[
  t = x \rightarrow t[1/x], t[0/x] \quad \text{(with } x \rightarrow y_0, y_1 = (x \land y_0) \lor (\neg x \land y_1) \text{)}
  \]

**Formula as decision tree**

Decision tree of \((x_1 \leftrightarrow y_1) \land (x_2 \leftrightarrow y_2)\) with the order in the variables \(x_1 < y_1 < x_2 < y_2\)
Symbolic model checking with BDD
Model checking with partial order reduction
Model checking with symmetry reduction
Bounded model checking

**Formula as BDD**

**BDD of** \((x_1 \leftrightarrow y_1) \land (x_2 \leftrightarrow y_2)\) **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** \(x_1 \lor x_3\) **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

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
  - $\text{var}(v) \in X$
  - $\text{low}(v) \in V$
  - $\text{high}(v) \in V$
- $\forall u, v \in V$ with $v = \text{low}(u)$ or $v = \text{high}(u) : \text{var}(u) < \text{var}(v)$
- $\text{var}(u) = \text{var}(v), \text{low}(u) = \text{low}(v), \text{high}(u) = \text{high}(v)$ implies $u = v$
- $\text{low}(u) \neq \text{high}(u)$
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.

**Constructing and manipulating ROBDDs**

ROBDD for \(x_1 \leftrightarrow y_1 \land x_2 \leftrightarrow y_2\) with the order \(x_1 < x_2 < y_1 < y_2\)

ROBDD for \(x_1 \leftrightarrow x_2 \land x_3 \leftrightarrow x_4\) with the order \(x_1 < x_2 < x_3 < x_4\)

Implementation with an array

<table>
<thead>
<tr>
<th>(u)</th>
<th>(var)</th>
<th>(low)</th>
<th>(high)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>3</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>5</td>
<td>2</td>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>2</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>7</td>
<td>1</td>
<td>5</td>
<td>6</td>
</tr>
</tbody>
</table>
Building a ROBDD

**Makenode**(\(H, \text{max}, b, i, \ell, h\)) : adding a node if it does not exist yet

Requirement for efficiency
- a hash table \(H : (i, \ell, h) \mapsto u\);
- \(\text{member}(H, i, \ell, h)\) true iff \((i, \ell, h) \in H\):
- \(\text{lookup}(H, i, \ell, h)\) return the position \(u\) of \((i, \ell, h)\) in \(b\);
- \(\text{insert}(H, i, \ell, h, u)\) insert in \(H\) for \((i, \ell, h)\) its position \(u\) in \(b\)

---

**Makenode**(\(H, \text{max}, b, i, \ell, h\)) : returns the “good” node (added if needed)

**Require:** \(H : (i, \ell, h) \mapsto u\),
**Require:** \(b\) the BDD in construction,
**Require:** \(\text{max}\) its current size,
**Ensure:** adds \((i, \ell, h)\) in \(b\) if needed and returns its position in \(b\)

if \(\ell = h\) then
  return \(\ell\)
else if \(\text{member}(H, i, \ell, h)\) then
  return \(\text{lookup}(H, i, \ell, h)\)
else
  \(\text{max} \leftarrow \text{max} + 1\)
  \(b.\text{var}(\text{max}) \leftarrow i\)
  \(b.\text{low}(\text{max}) \leftarrow \ell\)
  \(b.\text{high}(\text{max}) \leftarrow h\)
  \(\text{insert}(H, i, \ell, h, \text{max})\)
  return \(\text{max}\)
end if
**Build** maps a boolean expression \( t \) into a ROBDD

**Ensure:** build in \( b \) the ROBDD for \( t \) (Depth first and construction in postorder)

```plaintext
function build'(t, i) =
  if i > n then
    if t = ⊥ 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, \( \ell \), h) \{Builds node (if needed)\}
  end if
end if
end {build'}
```

\( H \leftarrow emptytable \)

\( max \leftarrow 1 \)

\( b.root \leftarrow build'(t, 1) \)

return \( b \)

---

**Build Example**

**Build** \( x_1 \leftrightarrow x_2 \)

![ROBDD Diagram](image-url)
Operations on ROBDD $R = A \ op \ B$

All binary operators are implemented by the same general algorithm

\[ \text{APPLY} (\ op, \ u_1, \ u_2) \ where \]

- \( \ op \) 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. \( x_i < x_j : (x_i \rightarrow h_1, \ell_1) \ op \ (x_j \rightarrow h_2, \ell_2) = (x_j \rightarrow (h_1 \ op \ (x_j \rightarrow h_2, \ell_2)), (\ell_1 \ op \ (x_j \rightarrow h_2, \ell_2)) \)
3. \( x_i > x_j : \) symmetric to case 2
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 \(\text{var}(u_1) = \text{var}(u_2)\) then
res \(\leftarrow \text{makenode}(\text{var}(u_1), \text{app}(\text{low}(u_1), \text{low}(u_2)), \text{app}(\text{high}(u_1), \text{high}(u_2)))\)
else if \(\text{var}(u_1) < \text{var}(u_2)\) then
res \(\leftarrow \text{makenode}(\text{var}(u_1), \text{app}(\text{low}(u_1), u_2), \text{app}(\text{high}(u_1), u_2))\)
else \(\{\text{var}(u_1) > \text{var}(u_2)\}\)
res \(\leftarrow \text{makenode}(\text{var}(u_2), \text{app}(u_1, \text{low}(u_2)), \text{app}(u_1, \text{high}(u_2)))\)
end if
\(G(u_1, u_2) \leftarrow res\)
return res
end if

APPLY \((op, b_1, b_2)\) (end)

for all \(i \leq \max(b_1) \land j \leq \max(b_2)\) do
\(G(i,j) \leftarrow empty\)
end for
\(b.\text{root} \leftarrow \text{app}(b_1.\text{root}, b_2.\text{broot})\)
return \(b\)
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
return res(u)

Example of restrict

u[0/x_2]

\[
\begin{align*}
x_1 & \quad \Rightarrow \\
x_2 & \\
x_2 & \\
x_3 & \\
1 & \\
0 & \\
x_3 & \\
1 & \\
0 & \\
\end{align*}
\]
Operations on ROBDD

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

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

Model checking with BDD

Example of model checking with BDD

Transition relation $R$

<table>
<thead>
<tr>
<th></th>
<th>$b_0$</th>
<th>$b_1$</th>
<th>$b_0'$</th>
<th>$b_1'$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0 0 0 1 1 1</td>
<td>0 1 1 0 0 1</td>
<td>0 1 1 1 1 0</td>
<td>1 1 0 1 0 1</td>
</tr>
</tbody>
</table>
Model checking with BDD

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

$p = \neg b_0$

$b_0 \quad 0 \quad 0$

$b_1 \quad 0 \quad 1$

Model checking with BDD

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

$p = \neg b_0$

$b_0 \quad 0 \quad 0$

$b_1 \quad 0 \quad 1$
Example 2: set of states that reach a state where $p$ holds

$\exists \diamond p = \mu X.(p \lor \langle \rangle X)$

\[
p = \neg b_0
\]

<table>
<thead>
<tr>
<th>$X$</th>
<th>$p \lor \langle \rangle X$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\emptyset$</td>
<td>$p$</td>
</tr>
<tr>
<td>$p$</td>
<td>$p \lor \langle \rangle p$</td>
</tr>
<tr>
<td>$p \lor \langle \rangle p$</td>
<td>$p \lor \langle \rangle (p \lor \langle \rangle p) = T$</td>
</tr>
</tbody>
</table>

$X \mid p \lor \langle \rangle X$

\[
\begin{array}{c}
\begin{array}{c}
X \mid p \lor \langle \rangle X
\end{array}
\end{array}
\]

$X \mid p \lor \langle \rangle X$

<table>
<thead>
<tr>
<th>$X$</th>
<th>$p \lor \langle \rangle X$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\emptyset$</td>
<td>$p$</td>
</tr>
<tr>
<td>$p$</td>
<td>$p \lor \langle \rangle p$</td>
</tr>
<tr>
<td>$p \lor \langle \rangle p$</td>
<td>$p \lor \langle \rangle (p \lor \langle \rangle p) = T$</td>
</tr>
</tbody>
</table>

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

$p = \neg b_0$

$X \mid p \lor \langle \rangle X$

<table>
<thead>
<tr>
<th>$X$</th>
<th>$p \lor \langle \rangle X$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\emptyset$</td>
<td>$p$</td>
</tr>
<tr>
<td>$p$</td>
<td>$p \lor \langle \rangle p$</td>
</tr>
<tr>
<td>$p \lor \langle \rangle p$</td>
<td>$p \lor \langle \rangle (p \lor \langle \rangle p) = T$</td>
</tr>
</tbody>
</table>

$X \mid p \lor \langle \rangle X$

<table>
<thead>
<tr>
<th>$X$</th>
<th>$p \lor \langle \rangle X$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\emptyset$</td>
<td>$p$</td>
</tr>
<tr>
<td>$p$</td>
<td>$p \lor \langle \rangle p$</td>
</tr>
<tr>
<td>$p \lor \langle \rangle p$</td>
<td>$p \lor \langle \rangle (p \lor \langle \rangle p) = T$</td>
</tr>
</tbody>
</table>

$X \mid p \lor \langle \rangle X$

<table>
<thead>
<tr>
<th>$X$</th>
<th>$p \lor \langle \rangle X$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\emptyset$</td>
<td>$p$</td>
</tr>
<tr>
<td>$p$</td>
<td>$p \lor \langle \rangle p$</td>
</tr>
<tr>
<td>$p \lor \langle \rangle p$</td>
<td>$p \lor \langle \rangle (p \lor \langle \rangle p) = T$</td>
</tr>
</tbody>
</table>

$X \mid p \lor \langle \rangle X$

<table>
<thead>
<tr>
<th>$X$</th>
<th>$p \lor \langle \rangle X$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\emptyset$</td>
<td>$p$</td>
</tr>
<tr>
<td>$p$</td>
<td>$p \lor \langle \rangle p$</td>
</tr>
<tr>
<td>$p \lor \langle \rangle p$</td>
<td>$p \lor \langle \rangle (p \lor \langle \rangle p) = T$</td>
</tr>
</tbody>
</table>

$X \mid p \lor \langle \rangle X$

<table>
<thead>
<tr>
<th>$X$</th>
<th>$p \lor \langle \rangle X$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\emptyset$</td>
<td>$p$</td>
</tr>
<tr>
<td>$p$</td>
<td>$p \lor \langle \rangle p$</td>
</tr>
<tr>
<td>$p \lor \langle \rangle p$</td>
<td>$p \lor \langle \rangle (p \lor \langle \rangle p) = T$</td>
</tr>
</tbody>
</table>
Example: BDD of $\text{suc}_3^R(p)$

$p = \neg b_0$

$\begin{array}{c}
\begin{array}{c}
\text{0} \\
\text{1}
\end{array} \\
\begin{array}{c}
\text{0} \\
\text{1}
\end{array}
\end{array}$

$\text{suc}_3^R(p) = \exists v. R \land p$

$\begin{array}{c}
\begin{array}{c}
\text{0} \\
\text{1}
\end{array} \\
\begin{array}{c}
\text{0} \\
\text{1}
\end{array}
\end{array}$

$suc_3^R(p) = \exists v. R \land p$

$\begin{array}{c}
\begin{array}{c}
\text{0} \\
\text{1}
\end{array} \\
\begin{array}{c}
\text{0} \\
\text{1}
\end{array}
\end{array}$

$b_0$ 0 0

$b_1$ 0 1

$b'\!\!'$

$b_0$

$suc_3^R(p) = \exists v. R \land p$

$\begin{array}{c}
\begin{array}{c}
\text{0} \\
\text{1}
\end{array} \\
\begin{array}{c}
\text{0} \\
\text{1}
\end{array}
\end{array}$

$(\neg b_0 \land b_1) \lor b_0$

$\begin{array}{c|c}
\text{i} & \bigvee_{i=0..n} suc_i^3(p) \\
\hline
0 & \neg b_0 \\
1 & \top
\end{array}$

Checking safety properties

Nothing “BAD” can happen

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

$\bullet \text{suc}_3^*(I) \cap BAD = \emptyset$ (forward search)
Checking safety properties

Nothing BAD can happen

\[ \equiv \text{No bad states are reachable from an initial state} \]

\[ \text{pre}_I^*(BAD) \cap I = \emptyset \] (backward search)

Plan

1. Symbolic model checking with BDD

2. Model checking with partial order reduction

3. Model checking with symmetry reduction

4. Bounded model checking
Asynchronous computation and interleaving semantics

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

Example: 3 independent events (asynchronous systems) \( P = a \parallel b \parallel c \)

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
  - Independent events (e.g. assignments to local variables), or
  - Dependant events (e.g. assignments to shared variables, synchronizations, ...)

System = 2 concurrent processes

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

$\Rightarrow$ here each assignment is a "transition": e.g. $\langle P_{10}, P_{20} \rangle \rightarrow \langle P_{11}, P_{20} \rangle$ and $\langle P_{10}, P_{21} \rangle \rightarrow \langle P_{11}, P_{21} \rangle$ 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

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

$\Rightarrow a, b, c, d, e$ 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
⇒ We extend the definition of Kripke structure

Definition : state transition system
\[ \mathcal{K} = \langle \mathcal{I}, S, \mathcal{T}, \mathcal{L} \rangle \] where
- \( \mathcal{T} \) is the set of transitions \( \alpha \subseteq S \times S \)
- \( \mathcal{I}, S, \mathcal{L} \) 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

- A transition \( \alpha \) is enabled in a state \( s \) if there is a state \( s' \) such that \( \alpha(s, s') \) holds
- Otherwise \( \alpha \) is disabled in \( s \).
- \( enabled(s) \) : set of transitions enabled in \( s \)
- A transition \( \alpha \) is deterministic if for every state \( s \), there is at most one \( s' \) such that \( \alpha(s, s') \)
- When \( \alpha \) is deterministic we write \( s' = \alpha(s) \)
- A path is a finite or infinite sequence
  \[ \pi = s_0 \xrightarrow{\alpha_0} s_1 \xrightarrow{\alpha_1} \ldots \]
  such that for every \( i : \alpha_i(s_i, s_{i+1}) \) holds.
- Any prefix of a path is a path
- \( |\pi| \) (length of \( \pi \)) : number of transitions in \( \pi \)

Note : we only consider deterministic transitions (this is natural)
Depth first search algorithm

1. $\text{hash}(s_0)$;
2. $\text{set on-stack}(s_0)$;
3. $\text{expand-state}(s_0)$;

4. procedure $\text{expand-state}(s)$
5. $\text{work-set}(s) := \text{ample}(s)$;
6. while $\text{work-set}(s)$ is not empty do
7. let $\alpha \in \text{work-set}(s)$;
8. $\text{work-set}(s) := \text{work-set}(s) \setminus \{\alpha\}$;
9. $s' := \alpha(s)$;
10. if new($s'$) then
11. $\text{hash}(s')$
12. $\text{set on-stack}(s')$
13. $\text{expand-state}(s')$
14. end if;
15. $\text{create-edge}(s, \alpha, s')$
16. end while;
17. $\text{set completed}(s)$;
18. end procedure

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

Principle of the algorithm

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

- When \( \text{ample}(s) \) is used instead of \( \text{enable}(s) \), enough behaviors must be retained so that DFS gives correct results.
- Using \( \text{ample}(s) \) instead of \( \text{enable}(s) \) should result in a significantly smaller state graph.
- The overhead in calculating \( \text{ample}(s) \) must be reasonably small.
Definitions

- An independence relation $I \subseteq T \times T$ is a symmetric, antireflexive relation such that for $s \in S$ 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 $D = (T \times T) \setminus I$

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 independent transitions**

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

- **Problem 1** : The checked property is sensible of the order between $\alpha$ and $\beta$
- **Problem 2** : choosing e.g. $\alpha$ first can enable new transitions not possible through the path with $\beta$ first
Visible and invisible transition

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

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

Stuttering equivalence

Two infinite paths $\sigma = s_0 \xrightarrow{\alpha_0} s_1 \xrightarrow{\alpha_1} \ldots$ and $\rho = r_0 \xrightarrow{\beta_0} r_1 \xrightarrow{\beta_1} \ldots$ are **stuttering equivalent** ($\sigma \sim_{st} \rho$) if there are two infinite sequences of integers
\[ 0 = i_0 < i_1 < i_2 < \ldots \text{ and } 0 = j_0 < j_1 < j_2 < \ldots \]
such that for every $k \geq 0$

- $L(s_k) = L(s_{k+1}) = \cdots = L(s_{k+1-1}) = L(r_k) = L(r_{k+1}) = \cdots = L(r_{k+1-1})$

Can also be defined for finite paths
Stuttering Equivalence

Stuttering equivalence example

Suppose \( \alpha \) 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 \( \phi \) invariant under stuttering
if and only if, for each pair of paths \( \pi \) and \( \pi' \) 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 $K_1$ and $K_2$ and suppose they have initial state resp. $s_0$ and $s'_0$.

$K_1$ is stuttering equivalent to $K_2$ if and only if

- For each path $\sigma$ of $K_1$ which starts in $s_0$, there is a path $\sigma'$ of $K_2$ starting in $s'_0$ such that $\sigma \sim_{st} \sigma'$
- For each path $\sigma'$ of $K_2$ which starts in $s'_0$, there is a path $\sigma$ of $K_1$ starting in $s_0$ such that $\sigma \sim_{st} \sigma'$

**Corollary**

For two stuttering equivalent transition systems $K_1$ and $K_2$ (with initial states resp. $s_0$ and $s'_0$) and every LTL$_X$ property $\phi$

\[
K_1 \models \phi \iff K_2 \models \phi
\]

**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

\[
\text{ample}(s) = \text{enabled}(s)
\]
Four conditions for selecting \( ample(s) \) to preserve satisfaction of \( \text{LTL}_x \) formulae

**Condition C0**

\[
ample(s) = \emptyset \iff enables(s) = \emptyset
\]

**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 \( \text{enabled}(s) \setminus \text{ample}(s) \) are all independent of those in \( \text{ample}(s) \).

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 \( \alpha \in \text{ample}(s) \) and each \( \beta_i \) is independent of all transitions in \( \text{ample}(s) \) including \( \alpha \).
2. An infinite sequence of \( \beta_0 \beta_1 \ldots \) where each \( \beta_i \) is independent of all transition in \( \text{ample}(s) \).

Construction of ample sets

- Since \( \beta_i \) are independent from \( \alpha \) they do not disable it.
- In particular for case 1, we have

```
We want paths with \( \alpha \) first stuttering equivalent to the one with \( \alpha \) last.
```
**Construction of ample sets**

**Condition C2 (invisibility)**

If $s$ is not fully expanded then every $\alpha \in \text{ample}(s)$ is invisible.

For paths of the form $\beta_0 \beta_1 \ldots$ that starts at $s$ with no $\beta_i$ in $\text{ample}(s)$

$\alpha \beta_0 \beta_1 \ldots$ 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.

- $\beta$ visible (change a proposition $p$)
- $\beta$ independent from invisible $\alpha_i$

![Diagram]

The process on the right performs the invisible $\alpha_i$ forever!
Construction of ample sets

Condition C3 (Cycle closing condition)

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

Potential problems

- **Problem 1**: The checked property is sensible of the order between $\alpha$ and $\beta$
- **Problem 2**: choosing e.g. $\alpha$ first can enable new transitions not possible through the path with $\beta$ first
Potential problems

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

Analysis of potential problem 1

- Assume ample(s) = {β}
- and s₁ is not in the reduced graph
- By condition C2, β must be invisible
  \[ s S_2 r \sim_{st} S_{1} r \]
- 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 s₁
- γ is independent of β. Otherwise, the sequence α β violates C1
- Then, γ is enabled in r
- Assume s₁ \( \xrightarrow{\gamma} s' \) and r \( \xrightarrow{\gamma} r' \)
- (β is invisible) s s₁ s'₁ \( \sim_{st} \) s s₂ 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 $P_i$
- $\text{pre}(\alpha):$ set of transitions whose execution may enable $\alpha$
- $\text{dep}(\alpha):$ set of transitions dependent of $\alpha$
- $T_i:$ set of transitions of process $P_i$
- $T_i(s) = T_i \cap \text{enabled}(s):$ set of transitions of process $P_i$ enabled in $s$
- $\text{current}_i(s):$ set of transitions of process $P_i$ enabled in some $s'$ such that $pc_i(s') = pc_i(s)$

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 empty \( T_i(s) \)
- Check whether \( ample(s) = T_i(s) \) satisfies condition C1 (see below)
- If \( T_i(s) \) is not good : take another non empty \( T_i(s) \) (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 \( \alpha \), interdependent to transitions in \( T_i(s) \), is enabled.
Possible causes
- \( \alpha \) belongs to process \( P_j \) with \( (j \neq i) \) : a necessary condition is that \( \text{dep}(T_i(s)) \cap T_j \neq \emptyset \) (can be checked effectively)
- \( \alpha \) (the first transition that violates C1) belongs to process \( P_i \) (\( \alpha \in T_i \))
  - Suppose \( \alpha \) 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 \( \alpha \in \text{current}_i(s) \)
  - \( \alpha \notin T_i(s) \)
  - \( \alpha \in \text{current}_i(s) \setminus T_i(s) \)
  - (\( \alpha \) disabled in \( s \); enabled in \( s' \)) : \( \exists \beta \in \text{pre}(\alpha) \) in the sequence from \( s \) to \( s' \)
  - Thus, a necessary condition is that \( \exists j \neq i. \text{pre}(\text{current}_i(s) \setminus T_i(s)) \cap T_j \neq \emptyset \) (can be checked effectively)
Symbolic model checking with BDD

Model checking with partial order reduction

Model checking with symmetry reduction

Bounded model checking

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
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 ↦ y)
- Given sets: explicitly enumerated
- Deferred sets: elements not given a priori

Relations, functions, ...
- \( \text{dom}(x), \text{ran}(x), \text{image } (r[S]), \), inverse \( (R^{-1}) \), composition \( (R0 ; R1) \), restrictions \( (U < R, U \equiv R, R > U, R \gg U) \), ...
- Partial function \( (x \mapsto y) \), total function \( (x \rightarrow y) \), injection \( (\leftrightarrow, \leftarrow) \), surjection \( (\leftrightarrow, \rightarrow) \), bijection \( (\rightarrow) \)
- Sequences, records, trees, ...

Operations
- Transform the state of a machine
- Must preserve the invariant
A simple login

**MACHINE** LoginVerySimple

**SETS** Session

**VARIABLES** active

**INVARIANT** active ⊆ Session

**INITIALISATION** active := Ø

**OPERATIONS**

```plaintext
res ← Login = ANY s WHERE s ∈ Session ∧ s ∉ active THEN
  res := s || active := active ∪ {s} END;
Logout(s) = PRE s ∈ active THEN
  active := active – {s} END
END
```

**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)

**In practice to be efficient**

- On the fly analysis
- Does not keep the state space
Dining Philosopher (without protocol)

MACHINE Philosophers
SETS Phil: Forks
CONSTANTS lFork, rFork

PROPERTIES
lFork ∈ Phil ⇔ Forks ∧ rFork ∈ Phil ⇔ Forks ∧
\(\text{card}(\text{Phil}) = \text{card}(\text{Forks})\) ∧ \(\forall pp. (pp \in \text{Phil} \Rightarrow \text{lFork}(pp) \neq \text{rFork}(pp))\) ∧
\(\forall st. (st \subset \text{Phil} \land st \neq \emptyset \Rightarrow \text{rFork}^{-1}[\text{lFork}(st)] \neq st)\)

VARIABLES taken

INVARIANT
\(\text{taken} \in \text{Forks} \Rightarrow \text{Phil} \land\)
\(\forall xx. (xx \in \text{dom}(\text{taken}) \Rightarrow (\text{lFork}(\text{taken}(xx)) = xx \lor \text{rFork}(\text{taken}(xx)) = xx))\)

INITIALISATION taken := ∅

OPERATIONS
TakeLeftFork(p, f) =
\(\text{PRE} p \in \text{Phil} \land f \in \text{Forks} \land f \notin \text{dom}(\text{taken}) \land \text{lFork}(p) = f\)
\(\text{THEN} \text{taken}(f) := p \text{ END};\)

TakeRightFork(p, f) =
\(\text{PRE} p \in \text{Phil} \land f \in \text{Forks} \land f \notin \text{dom}(\text{taken}) \land \text{rFork}(p) = f\)
\(\text{THEN} \text{taken}(f) := p \text{ END};\)

DropFork(p, f) =
\(\text{PRE} p \in \text{Phil} \land f \in \text{Forks} \land f \in \text{dom}(\text{taken}) \land \text{taken}(f) = p\)
\(\text{THEN} \text{taken} := f \notin \text{taken} \text{ END}\)

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 a 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

- **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*

**Current symmetry methods**

**Example of isomorphic graphs**

<table>
<thead>
<tr>
<th>a</th>
<th>g</th>
</tr>
</thead>
<tbody>
<tr>
<td>b</td>
<td>h</td>
</tr>
<tr>
<td>c</td>
<td>i</td>
</tr>
<tr>
<td>d</td>
<td>j</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>4</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>8</td>
</tr>
</tbody>
</table>

**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

![Diagram](image)

**Canonical form of a graph**

coding the graph

<table>
<thead>
<tr>
<th>Adjacency matrix</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0 0 1 1 1 1 0</td>
</tr>
<tr>
<td>0 0 0 1 1 0 1 0 1</td>
</tr>
<tr>
<td>0 0 0 0 1 0 1 1 0</td>
</tr>
<tr>
<td>0 0 0 0 1 1 1 1 0</td>
</tr>
<tr>
<td>1 1 1 0 0 0 0 0 0</td>
</tr>
<tr>
<td>1 1 0 1 0 0 0 0 0</td>
</tr>
<tr>
<td>1 0 1 1 0 0 0 0 0</td>
</tr>
<tr>
<td>0 1 1 1 0 0 0 0 0</td>
</tr>
</tbody>
</table>

\[ G = 000011100000110100001011000001111100000110100001011000001110000 \]

- 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 $\rightarrow$ 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
- ...

⇒ 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
- ⇒ 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 $x, y$

<table>
<thead>
<tr>
<th>State</th>
<th>Markers</th>
<th>Condition</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>$s_1 = \langle{d_1 \mapsto 0}, {d_1}\rangle$, $s_3 = \langle{d_2 \mapsto 0}, {d_2}\rangle$</td>
<td>$m(s_1) = m(s_3)$</td>
<td>$\checkmark$</td>
<td></td>
</tr>
<tr>
<td>$s_1 = \langle{d_1 \mapsto 0}, {d_1}\rangle$, $s_2 = \langle{d_2 \mapsto 0}, {d_1}\rangle$</td>
<td>$m(s_1) \neq m(s_2)$</td>
<td>$\checkmark$</td>
<td></td>
</tr>
<tr>
<td>$s_4 = \langle{d_1 \mapsto 1, d_2 \mapsto 2}, {d_1 \mapsto 1, d_2 \mapsto 2}\rangle$</td>
<td>$m(s_4) \neq m(s_5)$</td>
<td>$\checkmark$</td>
<td></td>
</tr>
</tbody>
</table>

Example of what our markers can and cannot distinguish

Two dining philosophers

<table>
<thead>
<tr>
<th>State</th>
<th>Markers</th>
<th>Condition</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>$s_1 = p_1\ \text{leftTakes}(f)$, $s_2 = p_1\ \text{rightTakes}(f)$</td>
<td>$m(s_1) \neq m(s_2)$</td>
<td>$\checkmark$</td>
<td></td>
</tr>
<tr>
<td>Each has one fork (see figures)</td>
<td>$m(s_1) = m(s_2)$</td>
<td>$\times$</td>
<td></td>
</tr>
</tbody>
</table>
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 \( \text{NonSym} \) and all \( y_i \) are elements of some deferred set \( D_j \), or all \( x_i \) 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

<table>
<thead>
<tr>
<th>Machine</th>
<th>Card</th>
<th>Model Checking Time</th>
<th>Number of Nodes</th>
<th>Speedup over</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>wo flood markers</td>
<td>wo flood markers</td>
<td></td>
</tr>
<tr>
<td>Russian</td>
<td>1</td>
<td>0.05 0.05</td>
<td>15 15</td>
<td>1.04</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>0.32 0.21</td>
<td>81 48</td>
<td>1.51</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>1.32 0.46</td>
<td>441 119</td>
<td>3.92</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>8.73 1.90</td>
<td>2325 248</td>
<td>9.81</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>54.06 12.18</td>
<td>11985 459</td>
<td>26.35</td>
</tr>
<tr>
<td>scheduler0</td>
<td>1</td>
<td>0.01 0.01</td>
<td>6 6</td>
<td>1.09</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>0.07 0.05</td>
<td>16 10</td>
<td>1.59</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>0.28 0.07</td>
<td>55 17</td>
<td>4.60</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>0.98 0.20</td>
<td>190 26</td>
<td>7.15</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>4.52 0.75</td>
<td>649 37</td>
<td>16.87</td>
</tr>
<tr>
<td></td>
<td>6</td>
<td>20.35 4.74</td>
<td>2188 50</td>
<td>42.60</td>
</tr>
<tr>
<td></td>
<td>7</td>
<td>114.71 43.47</td>
<td>7291 65</td>
<td>143.61</td>
</tr>
<tr>
<td>scheduler1</td>
<td>1</td>
<td>0.01 0.01</td>
<td>6 6</td>
<td>1.09</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>0.07 0.05</td>
<td>16 10</td>
<td>1.59</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>0.28 0.07</td>
<td>55 17</td>
<td>4.60</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>0.98 0.20</td>
<td>190 26</td>
<td>7.15</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>4.52 0.75</td>
<td>649 37</td>
<td>16.87</td>
</tr>
<tr>
<td></td>
<td>6</td>
<td>20.35 4.74</td>
<td>2188 50</td>
<td>42.60</td>
</tr>
<tr>
<td></td>
<td>7</td>
<td>114.71 43.47</td>
<td>7291 65</td>
<td>143.61</td>
</tr>
<tr>
<td>Peterson</td>
<td>1</td>
<td>0.28 0.28</td>
<td>49 27</td>
<td>1.87</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>0.01 0.01</td>
<td>6 6</td>
<td>1.09</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>0.41 0.11</td>
<td>145 29</td>
<td>4.50</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>2.96 0.34</td>
<td>825 51</td>
<td>16.62</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>23.93 1.70</td>
<td>5201 81</td>
<td>64.24</td>
</tr>
<tr>
<td></td>
<td>6</td>
<td>192.97 13.37</td>
<td>37009 120</td>
<td>275.75</td>
</tr>
<tr>
<td></td>
<td>7</td>
<td>941.46 167.95</td>
<td>297473 169</td>
<td>771.39</td>
</tr>
<tr>
<td>Philosophers</td>
<td>2</td>
<td>0.01 0.01</td>
<td>5 5</td>
<td>1.09</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>0.41 0.11</td>
<td>145 29</td>
<td>4.50</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>2.96 0.34</td>
<td>825 51</td>
<td>16.62</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>23.93 1.70</td>
<td>5201 81</td>
<td>64.24</td>
</tr>
<tr>
<td></td>
<td>6</td>
<td>192.97 13.37</td>
<td>37009 120</td>
<td>275.75</td>
</tr>
<tr>
<td></td>
<td>7</td>
<td>941.46 167.95</td>
<td>297473 169</td>
<td>771.39</td>
</tr>
<tr>
<td>Philosophers</td>
<td>2</td>
<td>0.11 0.05</td>
<td>21 8</td>
<td>3.02</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>1.56 0.15</td>
<td>337 13</td>
<td>28.83</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>123.64 5.99</td>
<td>11809 26</td>
<td>799.36</td>
</tr>
<tr>
<td>Towns 2</td>
<td>1</td>
<td>0.01 0.01</td>
<td>5 5</td>
<td>1.09</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>0.37 0.33</td>
<td>17 11</td>
<td>1.08</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>63.95 12.78</td>
<td>513 105</td>
<td>4.94</td>
</tr>
<tr>
<td>USB 1</td>
<td>1</td>
<td>0.21 0.20</td>
<td>29 29</td>
<td>0.96</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>8.42 4.74</td>
<td>694 355</td>
<td>1.36</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>605.25 277.59</td>
<td>16906 3013</td>
<td>2.60</td>
</tr>
</tbody>
</table>
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]

⇒ 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

⇒ To our knowledge, the first elaborate approach and evaluation of an efficient approximation method.
Beyond BDDs

- BDDs are still too large
- Variable order must be uniform along all considered paths
- Need to find “right” ordering
Beyond BDDs

BDDs are still too large  
variable order must be uniform along all considered paths  
need to find “right” ordering

idea

- use symbolic encoding by propositional formula
- rely on highly optimized SAT solver to do dfs-exploration

Boolean formula that is satisfied, if the underlying transition system realizes a finite trace that reaches a given set of states

Bounded Reachability

given a model $\mathcal{M} = \langle S, R, L \rangle$ over $AP$

we define the following predicate on states

- $Reach(s, s')$ iff $R(s, s')$
Bounded Reachability

- given a model $\mathcal{M} = \langle S, R, L \rangle$ over $AP$
- we define the following predicate on states
  - $Reach(s, s')$ iff $R(s, s')$
- now $\mathcal{M}^k = \bigwedge_{i=0}^{k-1} Reach(s_i, s_{i+1})$

Suppose the property $\Psi$ we want to model check is that an invariant property $P(s)$ holds for every reachable state of the system.
$\neg \Psi = \bigvee_{i=0}^{k} \neg P(s_i)$
Bounded Model Checking Algorithm

For each model $\mathcal{M}$ and each LTL formula $\varphi$ there exists a $k \in \mathbb{N}$ such that if $\varphi$ is satisfied in $[\mathcal{M}]^k$ then $\mathcal{M} \models \varphi$.

- how to find this $k$?
- finding smallest $k$ is as hard as model checking itself 😞
**BMC: Completeness Threshold**

**Theorem:**
For each model $M$ and each LTL formula $\varphi$ there exists a $k \in \mathbb{N}$ such that if $\varphi$ is satisfied in $[M]^k$ then $M \models \varphi$.

- how to find this $k$ ?
- finding smallest $k$ is as hard as model checking itself 😞
- approximate it !
  - $ct$ is the (c)omputation (t)hreshold, the minimal $k$ needed to show a property
  - what would be a ct for $Gp$ formulae ?
  - what would be a ct for $Fp$ formulae ?

**BMC: Complexity ?**

- underlying SAT question is solvable in $O(k \times (|M| + |\varphi|))$
  (due to relying on fixpoint based translation)
underlying SAT question is solvable in $\mathcal{O}(k \times (|M| + |\varphi|))$
(due to relying on fixpoint based translation)

$\diamondsuit$ $k$ can be as large as the diameter of $M$, this can be exponential

thus SAT is ExpTime 😐
BMC: Complexity?

- underlying SAT question is solvable in $O(k \times (|\mathcal{M}| + |\varphi|))$
  (due to relying on fixpoint based translation)
- $k$ can be as large as the diameter of $\mathcal{M}$, this can be exponential
- thus SAT is $\text{ExpTime}$ 😞
- SAT-based BMC is at least $2\text{ExpTime}$ 😞 😞

BMC: Complexity?

- underlying SAT question is solvable in $O(k \times (|\mathcal{M}| + |\varphi|))$
  (due to relying on fixpoint based translation)
- $k$ can be as large as the diameter of $\mathcal{M}$, this can be exponential
- thus SAT is $\text{ExpTime}$ 😞
- SAT-based BMC is at least $2\text{ExpTime}$ 😞 😞
- automata-based approach would only be $\text{ExpTime}$ 😞 😞 😞
BMC: Complexity ?

- underlying SAT question is solvable in $\mathcal{O}(k \times (|\mathcal{M}| + |\varphi|))$
  (due to relying on fixpoint based translation)
- $k$ can be as large as the diameter of $\mathcal{M}$, this can be exponential
- thus SAT is ExpTime 😐
- SAT-based BMC is at least $2\text{ExpTime}$ 😐 😐
- automata-based approach would only be ExpTime 😐 😐 😐
- however SAT solvers today are extremely efficient for BMC and
  most errors can be already found with small $k$ 😊😊😊

can you give a reason for the latter ?
Chapter 8: Specification Languages and Formal Description Techniques

CSP

Formal methods

Example of formal methods (from Wikipedia)

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

- 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

### Events:
- **Internal** to the process it belongs to (often denoted $\tau$, $\epsilon$, or $i$).
- **External**: will take place in a synchronization with one or several other process(es).

### Operators:
- Parallel composition of processes.
- Sequentialization of interactions.
- Choice.
- Hiding of interaction points.
- Recursion or process replication.
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.

Note

This section is taken from Jeremy Martin’s PhD thesis
Syntax

\[
\text{Process} ::= \text{STOP} \mid \text{SKIP} \mid \\
\text{event} \rightarrow \text{Process} \mid \\
\text{Process} \cdot \text{Process} \mid \\
\text{Process} && \text{Process} \mid \\
\text{Process} \setminus \text{Process} \mid \\
\text{Process} \setminus \text{event} \mid \\
f(\text{Process}) \mid \\
\text{name} \mid \\
\mu \text{name} \cdot \text{Process}
\]

A vending machine example

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

\[
\text{VM} = \text{coin} \rightarrow ((\text{tea} \rightarrow \text{VM}) \square (\text{coin} \rightarrow \text{coffee} \rightarrow \text{VM}))
\]

\[
\text{TD} = (\text{coin} \rightarrow \text{tea} \rightarrow \text{TD}) \square (\text{coffee} \rightarrow \text{TD})
\]

\[
\text{SYSTEM} = \text{VM} \sqcup \{(\text{coin,coffee,tea}) \mid (\text{coin,coffee,tea})\} \text{ TD}
\]

\[
= \left( (\text{coin} \rightarrow ((\text{tea} \rightarrow \text{VM}) \square (\text{coin} \rightarrow \text{coffee} \rightarrow \text{VM}))) \right)
\]

\[
\sqcup \{(\text{coin,coffee,tea}) \mid (\text{coin,coffee,tea})\}
\]

\[
= \left( (\text{coin} \rightarrow \text{tea} \rightarrow \text{TD}) \square (\text{coffee} \rightarrow \text{TD}) \right)
\]

\[
\text{using law 1.22 with } X = \{\text{coin}\}, Y = \{\text{coin,coffee}\}, Z = \{\text{coin}\}
\]

\[
= \text{coin} \rightarrow \text{tea} \rightarrow (\text{VM} \sqcup \{(\text{coin,coffee,tea}) \mid (\text{coin,coffee,tea})\} \text{ TD})
\]

\[
\text{using law 1.22 with } X = \{\text{tea,coin}\}, Y = \{\text{tea}\}, Z = \{\text{tea}\}
\]

\[
= \text{coin} \rightarrow \text{tea} \rightarrow \text{SYSTEM}
\]
### Axiomatic laws

\[
\begin{align*}
\text{SKIP} ; P & = P ; \text{SKIP} = P \quad (1.1) \\
\text{STOP} ; P & = \text{STOP} \quad (1.2) \\
(P ; Q) ; R & = P ; (Q ; R) \quad (1.3) \\
(a \rightarrow P) ; Q & = a \rightarrow (P ; Q) \quad (1.4) \\
P \begin{bmatrix} A | B \end{bmatrix} Q & = Q \begin{bmatrix} B | A \end{bmatrix} P \quad (1.5) \\
P \begin{bmatrix} A | B \cup C \end{bmatrix} (Q \begin{bmatrix} B | C \end{bmatrix} R) & = (P \begin{bmatrix} A | B \end{bmatrix} Q) \begin{bmatrix} A \cup B | C \end{bmatrix} R \quad (1.6) \\
P \begin{bmatrix} \begin{array}{c} \text{avoid} \\
\end{array} \end{bmatrix} Q & = Q \begin{bmatrix} \text{avoid} \end{bmatrix} P \quad (1.7) \\
P \begin{bmatrix} \text{skip} \end{bmatrix} = P \quad (1.8) \\
P \begin{bmatrix} \text{avoid} \end{bmatrix} (P \begin{bmatrix} \text{avoid} \end{bmatrix} R) & = (P \begin{bmatrix} \text{avoid} \end{bmatrix} Q) \begin{bmatrix} \text{avoid} \end{bmatrix} R \quad (1.9) \\
P \begin{bmatrix} \text{begin} \end{bmatrix} = P \quad (1.10) \\
P \begin{bmatrix} \text{begin} \end{bmatrix} Q & = Q \begin{bmatrix} \text{begin} \end{bmatrix} P \quad (1.11) \\
P \begin{bmatrix} \text{begin} \end{bmatrix} (Q \begin{bmatrix} \text{begin} \end{bmatrix} R) & = (P \begin{bmatrix} \text{begin} \end{bmatrix} Q) \begin{bmatrix} \text{begin} \end{bmatrix} R \quad (1.12)
\end{align*}
\]

### Axiomatic laws (cont'd)

\[
\begin{align*}
P \begin{bmatrix} \text{begin} \end{bmatrix} P & = P \quad (1.13) \\
P \begin{bmatrix} \text{begin} \end{bmatrix} Q & = Q \begin{bmatrix} \text{begin} \end{bmatrix} P \quad (1.14) \\
P \begin{bmatrix} \text{begin} \end{bmatrix} (Q \begin{bmatrix} \text{begin} \end{bmatrix} R) & = (P \begin{bmatrix} \text{begin} \end{bmatrix} Q) \begin{bmatrix} \text{begin} \end{bmatrix} R \quad (1.15) \\
P \begin{bmatrix} A | B \end{bmatrix} (Q \begin{bmatrix} \text{begin} \end{bmatrix} R) & = (P \begin{bmatrix} A | B \end{bmatrix} Q) \begin{bmatrix} \text{begin} \end{bmatrix} (P \begin{bmatrix} A | B \end{bmatrix} R) \quad (1.16) \\
P \begin{bmatrix} \text{begin} \end{bmatrix} (Q \begin{bmatrix} \text{begin} \end{bmatrix} R) & = (P \begin{bmatrix} \text{begin} \end{bmatrix} Q) \begin{bmatrix} \text{begin} \end{bmatrix} (P \begin{bmatrix} \text{begin} \end{bmatrix} R) \quad (1.17) \\
P \begin{bmatrix} \text{begin} \end{bmatrix} (Q \begin{bmatrix} \text{begin} \end{bmatrix} R) & = (P \begin{bmatrix} \text{begin} \end{bmatrix} Q) \begin{bmatrix} \text{begin} \end{bmatrix} (P \begin{bmatrix} \text{begin} \end{bmatrix} R) \quad (1.18) \\
(x \rightarrow P) \begin{bmatrix} \text{begin} \end{bmatrix} (x \rightarrow Q) & = (x \rightarrow P) \begin{bmatrix} \text{begin} \end{bmatrix} (x \rightarrow Q) \quad (1.19) \\
& = x \rightarrow (P \begin{bmatrix} \text{begin} \end{bmatrix} Q) \\
P \begin{bmatrix} \text{begin} \end{bmatrix} \text{STOP} & = P \quad (1.20) \\
\begin{bmatrix} \text{begin} \end{bmatrix} x, Q \begin{bmatrix} x \rightarrow P \end{bmatrix} x & = \text{STOP} \quad (1.21)
\end{align*}
\]
Axiomatic laws (cont’d)

Let \( P = \bigcirc_{x \in X} x \rightarrow P_x \)
\( Q = \bigcirc_{y \in Y} y \rightarrow Q_y \)

Then \( P \left[ A \mid B \right] Q = \bigcirc_{z \in Z} z \rightarrow (P'_z \left[ A \mid B \right] Q'_z) \)

where \( P'_z = \begin{cases} P_z & \text{if } z \in X \\ P & \text{otherwise} \end{cases} \)

and \( Q'_z = \begin{cases} Q_z & \text{if } z \in Y \\ Q & \text{otherwise} \end{cases} \)

and \( Z = (X \cap Y) \cup (X - B) \cup (Y - A) \)

assuming \( X \subseteq A \quad \text{and} \quad Y \subseteq B \)
\( \bigcirc_{k:B} (b \rightarrow P_b) \bigg| \bigcirc_{c:C} (c \rightarrow Q_c) \) = \((\bigcirc_{k:B} (b \rightarrow (P_b \bigg| \bigcirc_{c:C} (c \rightarrow Q_c)))) \bigg| \) \((\bigcirc_{c:C} (c \rightarrow (Q_c \bigg| \bigcirc_{k:B} (b \rightarrow P_c))))\) \quad (1.22)

Axiomatic laws (cont’d)

\( \text{SKIP} \setminus x = \text{SKIP} \) \quad (1.24)
\( \text{STOP} \setminus x = \text{STOP} \) \quad (1.25)
\( (P \setminus x) \setminus y = (P \setminus y) \setminus x \) \quad (1.26)
\( (x \rightarrow P) \setminus x = P \setminus x \) \quad (1.27)
\( (x \rightarrow P) \setminus y = x \rightarrow (P \setminus y) \quad \text{if } x \neq y \) \quad (1.28)
\( (P ; Q) \setminus x = (P \setminus x) ; (Q \setminus x) \) \quad (1.29)
\( (P \left[ A \mid B \right] Q) \setminus x = P \left[ A \mid B - \{x\} \right] (Q \setminus x) \)
\( \text{if } x \notin A \) \quad (1.30)
\( (P \cap Q) \setminus x = (P \setminus x) \cap (Q \setminus x) \) \quad (1.31)
\( ((x \rightarrow P) \bigcirc (y \rightarrow Q)) \setminus x = (P \setminus x) \cap ((P \setminus x) \bigcirc (y \rightarrow (Q \setminus x))) \)
\( \text{if } x \neq y \) \quad (1.32)
Axiomatic laws (cont'd)

\[
\begin{align*}
    f(\text{STOP}) &= \text{STOP} \quad (1.33) \\
    f(e \rightarrow P) &= f(e) \rightarrow f(P) \quad (1.34) \\
    f(P; Q) &= f(P) ; f(Q) \quad \text{if } f^{-1}(\sqrt{\cdot}) = \{\sqrt{\cdot}\} \quad (1.35) \\
    f(P \parallel Q) &= f(P) \parallel f(Q) \quad (1.36) \\
    f(P \Box Q) &= f(P) \Box f(Q) \quad (1.37) \\
    f(P \triangle Q) &= f(P) \triangle f(Q) \quad (1.38) \\
    f(P \backslash f^{-1}(x)) &= f(P) \backslash x \quad (1.39)
\end{align*}
\]

Basic definitions (examples)

\[
\{(\text{coffee,coffee,coffee}), (\text{coin,tea})\} \subseteq \text{traces}(TD)
\]

- Catenation: \( s \sim t \)
  \[
  \langle s_1, s_2, \ldots, s_m \rangle \sim \langle t_1, t_2, \ldots, t_n \rangle = \langle s_1, \ldots, s_m, t_1, \ldots, t_n \rangle
  \]

- 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: \( \langle a, b \rangle^3 = \langle a, b, a, b \rangle \)

- 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: \( \text{merge}(s, t) \) the set of all possible interleavings of trace \( s \) with trace \( t \)
  \[
  \langle (\text{coin,tea,coin,tea,coin,coin}), \{\text{tea,coin}\} \rangle \in \text{failures}(VM)
  \]
  \[
  \emptyset \in \text{divergences}(CLOCK \backslash \text{tick})
  \]
Example of requested properties

\[ \forall (s, X) : \text{failures}(P). \quad s \downarrow \text{in} > s \downarrow \text{out} \implies \text{out} \notin X \]

Properties of the set of failures

1. \( (\emptyset, \{\} ) \in F \)
2. \( (s \xrightarrow{t}, \{\} ) \in F \implies (s, \{\} ) \in F \)
3. \( (s, Y ) \in F \land X \subseteq Y \implies (s, X ) \in F \)
4. \( (s, X ) \in F \land (\forall c \in Y .((s \xrightarrow{c}, \{\} ) \notin F)) \implies (s, X \cup Y ) \in F \)
5. \( (\forall Y \in p(X). (s, Y ) \in F ) \implies (s, X ) \in F \)
6. \( s \in D \land t \in \Sigma^* \implies s \xrightarrow{t} \in D \)
7. \( s \in D \land X \subseteq \Sigma \implies (s, X ) \in F \)
Failure-Divergence semantics

\[(F_1, D_1) \subseteq (F_2, D_2) \iff F_1 \supseteq F_2 \land D_1 \supseteq D_2\]

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)

\[
\begin{align*}
\text{failures}(\bot) &= \Sigma^* \times P \Sigma \\
\text{divergences}(\bot) &= \Sigma^*
\end{align*}
\]

Least fixpoint computation

\[
\mu X \bullet F(X) = \bigsqcup \{ F^n(\bot) \mid n \in \mathbb{N} \}
\]
Failure-Divergence semantics

\[
\text{divergences}(P \uplus Q) = \text{divergences}(P) \cup \text{divergences}(Q)
\]

\[
\text{failures}(P \uplus Q) = \left\{ (s, X) \mid (s, X) \in \text{failures}(P) \cap \text{failures}(Q) \vee s \neq \emptyset \land (s, X) \in \text{failures}(P) \cup \text{failures}(Q) \right\}
\]

\[
\cup \left\{ (s, X) \mid s \in \text{divergences}(P \uplus Q) \right\}
\]

\[
\text{divergences}(P \setminus x) = \left\{ (s \uplus (\Sigma - \{x\})) \setminus t \mid s \in \text{divergences}(P) \land (\forall n.s^n(x) \in \text{traces}(P)) \right\}
\]

\[
\text{failures}(P \setminus x) = \left\{ ((s \uplus (\Sigma - \{x\}), X), (s, X \cup \{x\}) \in \text{failures}(P) \right\}
\]

\[
\cup \left\{ (s, X) \mid s \in \text{divergences}(P \setminus \{x\}) \right\}
\]

\[
\text{divergences}(f(P)) = \{ f(s) \mid s \in \text{divergences}(P) \}
\]

\[
\text{failures}(f(P)) = \{ (f(s), X), (s, f^{-1}(X)) \in \text{failures}(P) \}
\]

\[
\cup \left\{ (s, X) \mid s \in \text{divergences}(f(P)) \right\}
\]

Operational semantics: example

LTS where the current process identifies the global state
**Operational semantics**

**Primitive processes:**

\[
\begin{align*}
\text{skip} & \xrightarrow{\cdot} \text{stop} \\
(a \to P) & \xrightarrow{a} P
\end{align*}
\]

**Prefix:**

\[
(a \to P) \xrightarrow{a} P
\]

**External choice:**

\[
\begin{align*}
(P \diamond Q) & \xrightarrow{a} P' & a \neq \tau \\
(Q \diamond Q') & \xrightarrow{a} Q' & a \neq \tau \\
(P' \diamond Q) & \xrightarrow{\tau} (P' \diamond Q)
\end{align*}
\]

**Operational semantics**

**Internal choice:**

\[
\begin{align*}
(P \cap Q) & \xrightarrow{\tau} P \\
(P \cap Q) & \xrightarrow{\tau} Q
\end{align*}
\]

**Sequential Composition:**

\[
\begin{align*}
(P \cdot Q) & \xrightarrow{a} (P' \cdot Q) & a \neq \tau \\
(P \cdot Q) & \xrightarrow{\tau} Q
\end{align*}
\]
Operational semantics

Parallel Composition:

\[
P \xrightarrow{a} P' \\
\frac{P \parallel [A \mid B] \quad Q \xrightarrow{a} P' \parallel [A \mid B]}{P \parallel [A \mid B] \quad Q \xrightarrow{a} P' \parallel [A \mid B]} \quad a \in (A - B - \{\sqrt{\}}) \cup \{\tau\}
\]

\[
Q \xrightarrow{a} Q' \\
\frac{Q \xrightarrow{a} Q'}{P \parallel [A \mid B] \quad Q' \xrightarrow{a} P' \parallel [A \mid B]} \quad a \in (B - A - \{\sqrt{\}}) \cup \{\tau\}
\]

\[
P \xrightarrow{a} P' \parallel Q' \\
\frac{P \parallel Q \xrightarrow{a} P' \parallel Q'}{P \parallel [A \mid B] \quad Q' \xrightarrow{a} P' \parallel [A \mid B] \quad Q'} \quad a \in (A \cap B) \cup \{\sqrt{\}}
\]

Interleaving:

\[
P \parallel Q \xrightarrow{a} P' \parallel Q' \\
\frac{P \parallel Q \xrightarrow{a} P' \parallel Q'}{P \parallel Q \xrightarrow{a} P' \parallel Q' \parallel Q'} \quad a \neq \sqrt{\}
\]

\[
Q \parallel Q' \xrightarrow{a} Q' \parallel Q' \\
\frac{Q \parallel Q' \xrightarrow{a} Q' \parallel Q'}{P \parallel Q \parallel Q' \xrightarrow{a} P' \parallel Q' \parallel Q'} \quad a \neq \sqrt{\}
\]

\[
P \parallel Q \xrightarrow{a} P' \parallel Q' \\
\frac{P \parallel Q \xrightarrow{a} P' \parallel Q'}{P \parallel Q \parallel Q' \xrightarrow{a} P' \parallel Q' \parallel Q'}
\]

Hiding:

\[
P \xrightarrow{a} P' \\
\frac{(P \setminus A) \xrightarrow{a} (P' \setminus A)}{P \xrightarrow{a} P' \setminus A} \quad a \in A \cup \{\tau\}
\]

\[
P \xrightarrow{a} P' \\
\frac{(P \setminus A) \xrightarrow{a} (P' \setminus A)}{P \xrightarrow{a} P' \setminus A} \quad a \notin A \cup \{\tau\}
\]

Alphabet Transformation:

\[
P \xrightarrow{a} P' \\
\frac{f(P) \xrightarrow{f(a)} f(P')}{f(P) \xrightarrow{\hat{f}} f(P')}
\]

Recursion:

\[
\mu X \parallel F(X) \xrightarrow{\hat{\tau}} F(\mu X \parallel F(X))
\]
### Extentions

- Parameterized processes
  
  \[
  \text{BUFF}(in, out) = in \rightarrow out \rightarrow \text{BUFF}(in, out)
  \]

- Type of a channel
  
  \[
  \text{type}(c) = \{v | c.v \in \Sigma\}
  \]

### Translation extended CSP into CSP

\[
(c!v \rightarrow P) = (c.v \rightarrow P)
\]

\[
(c?x \rightarrow P(x)) = \Box_{v:\text{type}(c)}(c.v \rightarrow P(v))
\]
Chapter 9: Testing

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

<table>
<thead>
<tr>
<th>Level of detail</th>
<th>Accessibility</th>
</tr>
</thead>
<tbody>
<tr>
<td>system</td>
<td>white box</td>
</tr>
<tr>
<td>integration</td>
<td>black box</td>
</tr>
<tr>
<td>module</td>
<td></td>
</tr>
<tr>
<td>unit</td>
<td></td>
</tr>
<tr>
<td>portability</td>
<td></td>
</tr>
<tr>
<td>maintainability</td>
<td></td>
</tr>
<tr>
<td>efficiency</td>
<td></td>
</tr>
<tr>
<td>usability</td>
<td></td>
</tr>
<tr>
<td>reliability</td>
<td></td>
</tr>
<tr>
<td>functionality</td>
<td></td>
</tr>
</tbody>
</table>

Characteristics
Automated Model-Based Testing

IUT conforms model

IUT passes tests

pass fail

test generation tool

test execution tool

model

IUT

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 \( \langle S, L, T, s_0 \rangle \)

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

*(Input-Enabled) Transition Systems*

---

**Correctness**

*Implementation Relation* \textit{ioco}

\[
i \overset{ioco} {\rightarrow} s \quad \overset{\text{def}}{=} \quad \forall \sigma \in \text{Straces}(s) : \quad \text{out}(i \text{ after } \sigma) \subseteq \text{out}(s \text{ after } \sigma)
\]

\textbf{Intuition:}

\textit{i oco}-conforms to \textit{s}, iff

- if \textit{i} produces output \textit{x} after trace \textit{\sigma},
  then \textit{s} can produce \textit{x} after \textit{\sigma}

- if \textit{i} cannot produce any output after trace \textit{\sigma},
  then \textit{s} cannot produce any output after \textit{\sigma} (quiescence \delta)
**Correctness**

Implementation Relation \( \text{ioco} \)

\[
i \text{ioco} s \overset{\text{def}}{=} \forall \alpha \in \text{Straces}(s) : \text{out}(i \text{ after } \alpha) \subseteq \text{out}(s \text{ after } \alpha)
\]

\[
p \overset{\delta}{\rightarrow} p = \forall !x \in L_U \cup \{t\} . \ p \overset{!x}{\rightarrow}
\]

\[
\text{Straces}(s) = \{ \alpha \in (L\cup\{\delta\})^* | s \overset{\alpha}{\rightarrow} \}
\]

\[
p \text{ after } \alpha = \{ p' \mid p \overset{\alpha}{\rightarrow} p' \}
\]

\[
\text{out}(P) = \{ !x \in L_U \mid p \overset{!x}{\rightarrow} , p \in P \} \cup \{ \delta \mid p \overset{\delta}{\rightarrow} p, p \in P \}
\]
Genealogy of ioco

Labelled Transition Systems

IOTS (IOA, IOSM, IOLTS)

Canonical Tester

Testing Equivalences
(Preorders)

Quiescent Trace Preorder

Refusal Equivalence
(Preorder)

Repetitive Quiescent
Trace Preorder
(Suspension Preorder)

model

test generation

tests

test execution

IUT

pass / fail

Involves:

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

model / specification
**Test Cases**

Model of a test case
= transition system :

- 'quiescence' label \( \theta \)
- tree-structured
- finite, deterministic
- final states pass and fail
- from each state \( = \) pass, fail :
  - either one input \(!a\)
  - or all outputs \(?x\) and \(\theta\)

\[\begin{align*}
  \text{Model of a test case} &= \text{transition system :} \\
  &\text{\quad - 'quiescence' label } \theta \\
  &\text{\quad - tree-structured} \\
  &\text{\quad - finite, deterministic} \\
  &\text{\quad - final states pass and fail} \\
  &\text{\quad - from each state } = \text{ pass, fail :} \\
  &\text{\quad \quad either one input } !a \\
  &\text{\quad \quad or all outputs } ?x \text{ and } \theta
\end{align*}\]

**\text{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 \( \varepsilon \):

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

1. **end test case**
   - pass

2. **supply input**
   - \(!a\)

3. **observe output**
   - forbidden outputs
     - \( ?y \)
     - fail
     - fail
   - allowed outputs
     - \(?x\)
     - \( T(S \text{ after } ?a = \varnothing) \)
   - allowed outputs or \( \delta \): \( ?x \in \text{out}(S) \)
   - forbidden outputs or \( \delta \): \( ly \notin \text{out}(S) \)
Test Generation Example

Test Execution Example

Two test runs:

1. `dub tea` -> `pass` || `i'`
2. `dub choc` -> `fail` || `i''`

i fails
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
  \[ i \text{ioco} s \implies i \text{passes } t \]

- **Exhaustiveness**: each incorrect implementation can be detected with a generated test \( t \)
  \[ i \text{ioco} s \implies \exists t : i \text{fails } t \]

Formal Testing with Transition Systems

Test assumption:
\[ \forall iUT \in \text{IMP}. \exists iUT \in \text{IOTS} . \]
\[ \forall t \in \text{TTS} . iUT \text{ passes } t \iff i\text{TUT passes } t \]

Proof soundness and exhaustiveness:
\[ \forall i\in \text{IOTS} . \]
\[ ( \forall t \in \text{gen}(s) . i \text{ passes } t ) \iff i \text{ioco } s \]
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 $\theta$
- tree-structured
- finite, deterministic
- final states pass and fail
- from each state $\triangleright$ pass, fail:
  - either one input $!a$
  - or all outputs $?x$ and $\theta$
Component Based Testing

Compositional Testing

Component Based Testing

If $s_1, s_2$ input enabled - $s_1, s_2 \in \text{IOTS}$ - then $\text{ioco}$ is preserved!
Underspecification: \textit{uioco}

\[ i \text{ ioco } s \iff \forall \sigma \in \text{Straces}(s) : \text{out}(i \text{ after } \sigma) \subseteq \text{out}(s_0 \text{ after } \sigma) \]

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

but \( ?b \notin \text{Straces}(s) : \) under-specification:

anything allowed after \( ?b \)

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

and \( ?a ?a \in \text{Straces}(s) \)

but from \( s_2 \), \( ?a ?a \) is under-specified:

anything allowed after \( ?a ?a \)

\[ \text{Utraces}(s) = \]

\[ \{ \sigma \in \text{Straces}(s) \mid \forall \sigma_1 ?a \sigma_2 = \sigma, \]

\[ \forall s' : s \xrightarrow{\sigma_1} s' \Rightarrow s' \xrightarrow{?a} \} \]

Now \( s \) is under-specified in \( s_2 \) for \( ?a \):

anything is allowed.

\[ \text{ioco} \subseteq \text{uioco} \]
Underspecification: \textit{uioco}

\[ i \text{ \textit{uioco} } s \iff \forall \sigma \in \text{Utraces}(s) : \text{out} \ (i \ \text{after} \ \sigma) \subseteq \text{out} \ (s_0 \ \text{after} \ \sigma) \]

Alternatively, via chaos process $\chi$ for under-specified inputs

Testing Components

<table>
<thead>
<tr>
<th>Method invocation</th>
<th>Method invocations</th>
<th>Method invocations</th>
</tr>
</thead>
<tbody>
<tr>
<td>IUT component</td>
<td>IUT component</td>
<td>IUT component</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Method invocation</th>
<th>Methods invoked</th>
<th>Method called</th>
<th>Method returned</th>
</tr>
</thead>
<tbody>
<tr>
<td>IUT component</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

© Jan Tretmans
Testing Components

$L_I = \text{offered methods calls } \cup \text{ used methods returns}$

$L_U = \text{offered methods returns } \cup \text{ used methods calls}$

Input-enabledness:

$\forall s \text{ of IUT}, \forall a \in L_I: s \overset{a}{\rightarrow}$

No ! ?
Correctness
Implementation Relation \textit{wioco}

\begin{align*}
i \text{uioco} s &= \text{def } \forall \alpha \in \text{Utraces}(s) : \text{out}(i \text{ after } \alpha) \subseteq \text{out}(s \text{ after } \alpha) \\
i \text{wioco} s &= \text{def } \forall \alpha \in \text{Utraces}(s) : \text{out}(i \text{ after } \alpha) \subseteq \text{out}(s \text{ after } \alpha) \\
&\quad\quad\quad\quad\text{and}\quad \text{in}(i \text{ after } \alpha) \supseteq \text{in}(s \text{ after } \alpha) \\
\text{in}(s \text{ after } \alpha) &= \{ \alpha \in L_I \mid s \text{ after } \alpha \text{ must } \alpha \} \\
s \text{ after } \alpha \text{ must } \alpha &= \forall s' (s \stackrel{\alpha}{\rightarrow} s' \Rightarrow s' \stackrel{\alpha}{\rightarrow})
\end{align*}

Variations on a Theme

\begin{align*}
i \text{ioco} s &\iff \forall \sigma \in \text{Straces}(s) : \text{out}(i \text{ after } \sigma) \subseteq \text{out}(s \text{ after } \sigma) \\
i \leq_{\text{i.or}} s &\iff \forall \sigma \in (L \cup \{\delta\})^* : \text{out}(i \text{ after } \sigma) \subseteq \text{out}(s \text{ after } \sigma) \\
i \text{iocnf} s &\iff \forall \sigma \in \text{traces}(s) : \text{out}(i \text{ after } \sigma) \subseteq \text{out}(s \text{ after } \sigma) \\
i \text{ioco}_{F} s &\iff \forall \sigma \in F : \text{out}(i \text{ after } \sigma) \subseteq \text{out}(s \text{ after } \sigma) \\
i \text{uioco} s &\iff \forall \sigma \in \text{Utraces}(s) : \text{out}(i \text{ after } \sigma) \subseteq \text{out}(s \text{ after } \sigma) \\
i \text{mioco} s &\quad\text{multi-channel ioco} \\
i \text{wioco} s &\quad\text{non-input-enabled ioco} \\
i \text{sioco} s &\quad\text{symbolic ioco} \\
i \text{(r)}\text{tioco} s &\quad\text{(real) timed ioco} \quad (\text{Aalborg, Twente, Grenoble, Bordeaux, ...}) \\
i \text{ioco}_{r} s &\quad\text{refinement ioco} \\
i \text{hioco} s &\quad\text{hybrid ioco}
\end{align*}
Implementation Relation \textit{ioco}

\[
i \text{ioco} s \overset{\text{def}}{=} \forall \sigma \in \text{Straces} (s) : \text{out} (i \text{ after } \sigma) \subseteq \text{out} (s \text{ after } \sigma)
\]

\[
\text{out} (i \text{ after } ?dub, ?dub) = \text{out} (s \text{ after } ?dub, ?dub) = \{ !\text{tea}, !\text{coffee} \}
\]

\[
\text{out} (i \text{ after } ?dub \delta, ?dub) = \{ !\text{coffee} \} = \text{out} (s \text{ after } ?dub \delta, ?dub) = \{ !\text{tea}, !\text{coffee} \}
\]
Variations on a Theme: mioco

\[ i \text{ mioco } s \iff \forall \sigma \in \text{Straces}'(s) : \text{out}'(i \text{ after } \sigma) \subseteq \text{out}'(s \text{ after } \sigma) \]

\[ \text{Straces}'(s) = \{ \sigma \in (L \cup \delta_k)^* | s \xrightarrow{\sigma} \} \]

\[ \text{out}'(P) = \{ |x| \in L_U | p \xrightarrow{x} , p \in P \} \cup \{ \delta_k | p \xrightarrow{\delta_k} p , p \in P \} \]

Testing Transition Systems: Extensions

\[ \frac{dV_t}{dt} = 3 \]

\[ [V_t = 15] \rightarrow \text{! tea} \]

\[ \frac{dV_c}{dt} = 2 \]

\[ [V_c \geq 5 \text{ID}] \rightarrow \text{! coffee} \]
Transition System with Data

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

Symbolic Data

STS → IOSTS

sioco

LTS

ioco

IOTS

test execution
test generation

Symbolic test generation

STTEST

TEST
Symbolic Transition System

Semantics $[S]$, of $STS = \langle L, l_0, V, I, \Lambda, \rightarrow \rangle$ in the context of initialisation $\nu \in U^V$ is

$LTS \langle L \times U^V, (l_0, \nu), \Sigma, \rightarrow \rangle$, where

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

\[
(l, \varphi, \rho) \vdash l' \quad \xi \in U^{\text{type}(\lambda)} \quad \vartheta \cup \xi \models \varphi \quad \vartheta' = (\vartheta \cup \xi)_{\text{eval}} \circ \rho \\
(l, \vartheta) \xrightarrow{(\lambda, \xi_{\text{type}(\lambda)})} (l', \vartheta')
\]
Symbolic Quiescence

Symbolic quiescence in location \( l_1 \):
\[
\Delta(l_1) = -\exists m : \text{int} . \ 0 < m < -v \\
\land -\exists m : \text{int} . \ 0 < m < v \\
= \ -1 \leq v \leq 1
\]

Symbolic suspension switch relation
\[
\text{out}!m : \text{int} \ [0 < m < -v] \\
\text{out}!m : \text{int} \ [0 < m < v]
\]

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\(_s\)

...... \(<\text{symbolic state}>\) after\(_s\) \(<\text{symbolic suspension trace}>\) ......

Lemma
\[
[[ <\text{symbolic state}>\text{ after}_s <\text{symbolic suspension trace}> ]] = [[[\text{symbolic state}]]\text{ after} [[[\text{symbolic suspension trace}]]]]
\]
Symbolic ioco

Specification: IOSTS $S(\iota_S) = (L_S, l_S, V_S, I, \Lambda, \rightarrow_S)$
Implementation: IOSTS $P(\iota_P) = (L_P, l_P, V_P, I, \Lambda, \rightarrow_P)$
both initialised, implementation input-enabled, $V_S \cap V_P = \emptyset$
$F_s$: a set of symbolic extended traces satisfying $[F_s]_{\iota_S} \subseteq Straces((l_0, \iota))$

$$P(\iota_P) \overset{sio} \rightarrow F_s S(\iota_S) \quad \text{iff} \quad \forall (\sigma, \chi) \in F_s \forall \lambda \in \Lambda_G \cup \{\delta\} : (\iota_P \cup \iota_S) \models \mathcal{F}(l_P, \lambda, \sigma) \land \chi \rightarrow \Phi(l_S, \lambda, \sigma)$$
where $\Phi(\xi, \lambda, \sigma) = \bigvee \{ \phi \land \psi \mid (\lambda, \varphi, \psi) \in \text{out}_s((\xi, \top, \text{id}) \text{after}_s(\sigma, \top))\}$

Theorem 1.

$$P(\iota_P) \overset{sio} \rightarrow F_s S(\iota_S) \quad \text{iff} \quad [P]_{\iota_P} \overset{sio} \rightarrow [F_s]_{\iota_S} [S]_{\iota_S}$$

An Application: Coverage

Location coverage
$l_0, l_1, l_2, l_3$

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

Symbolic state coverage
$(l_0, [\text{true}], v := v)$
$(l_1, [\text{true}], v := n_1)$
$(l_1, [-1 < n_1 < 1], v := n_1)$
$(l_2, [0 < m_2 < -n_1], v := n_1)$
$(l_3, [0 < m_2 < n_1], v := n_1)$
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:
  - ioca 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
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

Plan

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

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 $p$

- $\text{Var}(s)$: set of variables of $s$
- $\text{Free}(p)$: set of variables not bounded by a quantifier $\exists x$ or $\forall x$
Substitution $s \leftarrow u := t$

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

- 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

- 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 $T$.
- $\mathcal{D}_{\text{Integer}} = \{0, 1, -1, 2, -2, \ldots\}$;
- $\mathcal{D}_{\text{Boolean}} = \{\text{true}, \text{false}\}$;
- $\mathcal{D}_{T_1 \times \cdots \times T_n}^{T} = \mathcal{D}_{T_1} \times \cdots \times \mathcal{D}_{T_n} \Rightarrow \mathcal{D}_T$;
- The semantic domain
  \[ \mathcal{D} = \bigcup_T \text{ a type } \mathcal{D}_T \]}
Interpretation of variables

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

\[ \sigma : \text{Var} \rightarrow \mathcal{D} \]

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 \( \mathcal{D}_{T_1} \times \cdots \times \mathcal{D}_{T_n} \rightarrow \mathcal{D}_T \), 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

\[ \mathcal{I}[s] : \Sigma \rightarrow \mathcal{D} \]

- 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 \( f \):
  \[ \mathcal{I}[s](\sigma) = f(\mathcal{I}[s_1](\sigma), \ldots, \mathcal{I}[s_n](\sigma)) \]
Expression, substitution, proper state
Semantics and proof of a program
Proof system

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

\( I[s] : \Sigma \rightarrow D \)

- \( s \equiv a[s_1, \ldots, s_n] \) is an array:
  \[
  I[s](\sigma) = \sigma(a)[I[s_1](\sigma), \ldots, I[s_n](\sigma)]
  \]

- \( s \equiv \text{if } B \text{ then } s_1 \text{ else } s_2 \) : 
  \[
  I[s](\sigma) = \begin{cases} 
  I[s_1](\sigma) & \text{if } I[B](\sigma) = \text{true}; \\
  I[s_2](\sigma) & \text{if } I[B](\sigma) = \text{false}.
  \end{cases}
  \]

- \( s \equiv (s_1) \) :
  \[
  I[s](\sigma) = I[s_1](\sigma)
  \]

\( I[\cdot] \) is fixed for the constants: hence \( I[s](\sigma) \) is shorten by \( \sigma(s) \).

373

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 \( \sigma \);

- 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 \( \sigma \) to all variables except for \( a \):

  \[
  \sigma \ll u := d \gg (a)(d_1, \ldots, d_n) = \begin{cases} 
  d & \text{if } \land_i d_i = \sigma(t_i) \\
  \sigma(a)(d_1, \ldots, d_n) & \text{otherwise.}
  \end{cases}
  \]

374
Semantics of expressions and assertions (cont’d) : states defined by an assertion $p$

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

Some properties:
- $[\neg p] = \Sigma \setminus [p]$;
- $[p \lor q] = [p] \cup [q]$;
- $[p \land q] = [p] \cap [q]$;
- $p \rightarrow q \iff [p] \subseteq [q]$;
- $p \leftrightarrow q \iff [p] = [q]$.

Substitution lemma

The restriction of a proper state $\sigma$ to a subset of variables $X$, is denoted:

$$\sigma[X]$$

Lemma

For any assertion $p$, expressions $s$, $r$ and proper states $\sigma$ and $\tau$:
- If $\sigma[\text{Var}(s)] = \tau[\text{Var}(s)]$ then $\sigma(s) = \tau(s)$;
- If $\sigma[\text{Free}(p)] = \tau[\text{Free}(p)]$ then $\sigma \models p$ if $\tau \models p$.

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

The proofs are omitted here
Plan

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

Formal proof

Hoare triples
Correctness formula
\[
\{ p \} \quad P \quad \{ q \}
\]

Proof system:
- Axioms: "given formulas";
- Proof rules: used to establish "new" formulas from axioms or already established formulas.
Formal proof

Format of a rule

\[
\begin{array}{c}
\phi_1, \ldots, \phi_n \\
\hline
\psi
\end{array}
\]

where “…”

This rule says that \( \psi \) 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 \( \varphi_i \) 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 \( \psi \) can be established from the proof system \( P \).

A programming language and its formal semantics

Syntax

\[
S ::= \quad \text{skip} \\
| \ u := t \\
| \ S_1; S_2 \\
| \ \text{if } B \text{ then } S_1 \text{ else } S_2 \text{ fi} \\
| \ \text{while } B \text{ do } S_1 \text{ od}
\]

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

We suppose programs are well typed.

Abbreviation :

\[
\text{if } B \text{ then } S_1 \text{ fi}
\]

is the short for

\[
\text{if } B \text{ then } S_1 \text{ else skip fi}
\]
A programming language and its formal semantics

Semantics of a program

A program defines a function from initial states to final states:

\[ M[S] : \Sigma \rightarrow \Sigma \cup \{ \bot \} \]

where \( \bot \) denotes divergence.

Two approaches exist to define \( M[S] \):

- 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”**

\[ \langle S, \sigma \rangle \rightarrow \langle R, \tau \rangle \]

Execute \( S \) from the state \( \sigma \) produces the state \( \tau \) and \( R \) is the part of the program which remains to be executed.

Note: to express the termination, we can have \( R \equiv E \) (the empty program).

\( M[S] \) is based on the relation \( \rightarrow \) 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. $\langle \text{skip}, \sigma \rangle \rightarrow \langle E, \sigma \rangle$
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. $\langle \text{if } B \text{ then } S_1 \text{ else } S_2 \text{ fi}, \sigma \rangle \rightarrow \langle S_1, \sigma \rangle$ where $\sigma \models B$
5. $\langle \text{if } B \text{ then } S_1 \text{ else } S_2 \text{ fi}, \sigma \rangle \rightarrow \langle S_2, \sigma \rangle$ where $\sigma \not\models B$
6. $\langle \text{while } B \text{ do } S_1 \text{ od}, \sigma \rangle \rightarrow \langle S_1; \text{while } B \text{ do } S_1 \text{ od}, \sigma \rangle$ where $\sigma \models B$
7. $\langle \text{while } B \text{ do } S_1 \text{ od}, \sigma \rangle \rightarrow \langle E, \sigma \rangle$ 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.

### Definitions

- A **transitions sequence** of $S$ which starts in $\sigma$ is a finite or infinite sequence of configurations $\langle S_i, \sigma_i \rangle$ such that
  
  \[ \langle S, \sigma \rangle \rightarrow \langle S_1, \sigma_1 \rangle \rightarrow \cdots \rightarrow \langle S_i, \sigma_i \rangle \rightarrow \cdots \]

- An **execution** of $S$ which starts in $\sigma$ is a transition sequence from $\sigma$ which cannot be extended.

- An execution of $S$ which starts in $\sigma$ 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**:
  \[ M[S](\sigma) = \{ \tau | \langle S, \sigma \rangle \rightarrow^* \langle E, \tau \rangle \} \]

- **total correctness semantics**:
  \[ M_{\text{tot}}[S](\sigma) = \{ \tau | \langle S, \sigma \rangle \rightarrow^* \langle E, \tau \rangle \} \cup \{ \bot | S \text{ diverges from } \sigma \} \]

Some more notions:

- $\Omega \equiv \text{while } true \text{ do } \text{skip } \text{od}$, a never ending program;
- $(\text{while } B \text{ do } S \text{ od})^0 = \Omega$;
- $(\text{while } B \text{ do } S \text{ od})^{k+1} = \text{if } B \text{ then } S; (\text{while } B \text{ do } S \text{ od})^k \text{ else skip}$.
A programming language and its formal semantics

Semantic function

- $\mathcal{M}[S]$, $\mathcal{M}_{\text{tot}}[S]$ are extended to $\Sigma \cup \{\bot\}$ with:
  - $\mathcal{M}(\bot) = \emptyset$ and $\mathcal{M}_{\text{tot}}(\bot) = \{\bot\}$
- and to sets with:
  - $\mathcal{M}[S](X) = \bigcup_{\sigma \in X} \mathcal{M}[S](\sigma)$
  - $\mathcal{M}_{\text{tot}}[S](X) = \bigcup_{\sigma \in X} \mathcal{M}_{\text{tot}}[S](\sigma)$

Notation: $\mathcal{N}[S]$ stands for $\mathcal{M}[S]$ and $\mathcal{M}_{\text{tot}}[S]$

Some properties of semantic functions

- $\mathcal{N}[S]$ is monotone;
- $\mathcal{N}[S_1; S_2](X) = \mathcal{N}[S_2](\mathcal{N}[S_1](X))$;
- $\mathcal{N}[(S_1; S_2); S_3](X) = \mathcal{N}[S_1; (S_2; S_3)](X)$;
- $\mathcal{N}[[\text{if } B \text{ then } S_1 \text{ else } S_2 \text{ fi}]](X) = \mathcal{N}[S_1](X \cap [B]) \cup \mathcal{N}[S_2](X \cap [\neg B])$;
- $\mathcal{M}[[\text{while } B \text{ do } S_1 \text{ od}]] = \bigcup_{k=0}^{\infty} \mathcal{M}[[\text{while } B \text{ do } S_1 \text{ od}]^k]$
Definitions
For a program $S$
- $Var(S)$: variables used by $S$
- $Change(S)$: variables changed by $S$

Other properties
- For any proper states $\sigma$ and $\tau$, if $\tau \in \mathcal{N}[S](\sigma)$:
  $$\tau[Var \setminus Change(S)] = \sigma[Var \setminus Change(S)]$$
- For any proper states $\sigma$ and $\tau$ such that $\tau[Var(S)] = \sigma[Var(S)]$:
  $$\mathcal{N}[S](\sigma)[Var(S)] = \mathcal{N}[S](\tau)[Var(S)]$$

Definition of correct program

The correctness formula $\{p\} S \{q\}$ is true:
- for the partial correctness, denoted $\models \{p\} S \{q\}$, iff
  $$\mathcal{M}[S](\llbracket p \rrbracket) \subseteq \llbracket q \rrbracket$$
- for the total correctness, denoted $\models_{tot} \{p\} S \{q\}$, iff
  $$\mathcal{M}_{tot}[S](\llbracket p \rrbracket) \subseteq \llbracket q \rrbracket$$
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**  \( \text{Ax1} : \{p\} \text{skip}\{p\} \)
- **assignment**  \( \text{Ax2} : \{p \triangleleft u := t \} u := t\{p\} \)
- **Composition**  \( \text{R3} : \frac{\{p\} S_1\{r\}, \{r\} S_2\{q\}}{\{p\} S_1; S_2\{q\}} \)
- **Test**  \( \text{R4} : \frac{\{p \land B\} S_1\{q\}, \{p \land \neg B\} S_2\{q\}}{\{p\} \text{if } B \text{ then } S_1 \text{ else } S_2\{q\}} \)
- **Iteration**  \( \text{R5} : \frac{\{p \land B\} S\{p\}}{\{p\} \text{while } B \text{ do } S \text{ od}\{p \land \neg B\}} \)
- **consequence**  \( \text{R6} : \frac{\{p \rightarrow p_1, \{p_1\} S\{q_1\}, q_1 \rightarrow q\}}{\{p\} S\{q\}} \)

Example of partial correctness

For

\[
S \equiv x := 1; \\
a[x] := 2;
\]

do we have?

\[
\models \{\text{true}\} S\{a[1] = 2\}
\]
Proof system for the partial correctness

Example from Tony Hoare: the integer division

- specification:
  \[ \{ x \geq 0 \land y \geq 0 \} \]
  \[
  \text{DIV} \quad \{ quo \cdot y + rem = x \land 0 \leq rem < y \}
  \]

- the program \textsc{DIV}:
  \[
  quo := 0; \quad rem := x; \quad \\
  \text{while } rem \geq y \text{ do} \quad \text{rem} := \text{rem} - y; \quad \text{quo} := \text{quo} + 1; \quad \text{end while}
  \]

Proof system for the partial correctness

To establish:

\[ \{ x \geq 0 \land y \geq 0 \} \]
\[
\text{DIV} \quad \{ quo \cdot y + rem = x \land 0 \leq rem < y \}
\]

we have to find an invariant \( p \) such that:

- The invariant is verified when the while is reached:
  \[ \{ x \geq 0 \land y \geq 0 \} \]
  \[
  quo := 0; \quad rem := x; \quad \{ p \}
  \]

- The invariant remains true after each iteration:
  \[ \{ p \land B \} \]
  \[
  \text{rem} := \text{rem} - y; \quad \text{quo} := \text{quo} + 1 \quad \{ p \}
  \]

- When the condition in the while becomes false it implies the post-condition:
  \[ p \land \neg B \rightarrow quo \cdot y + rem = x \land 0 \leq rem < y \]

Find \( p : p \equiv quo \cdot y + rem = x \land rem \geq 0 \)

Generally the invariant is found by weakening the post-condition (it cannot be automatized).
Axioms and rules for the total correctness

Axioms and rules for the partial correctness (A1-R6), together with:

- iteration II R7:
  \[
  \begin{align*}
  \{p \land B\} & S\{p\}, \\
  \{p \land B \land t = z\} & S\{t < z\}, \\
  p & \rightarrow t \geq 0 \\
  \{p\} & \textbf{while } B \textbf{ do } S_1 \textbf{ od}\{p \land \neg B\}
  \end{align*}
  \]

$t$ is called termination function.

Total correctness Example: the integer division

Termination function?

\[ t \equiv \text{rem} \]

The invariant used for the partial correctness was too weak.

We need the following invariant for the proof of the total correctness:

\[ p' \equiv \text{quo} \cdot y + \text{rem} = x \land \text{rem} \geq 0 \land y > 0 \]
Partial and total correctness

Full example: Fibonacci’s suite

Definition of Fibonacci’s suite:
\[ F_0 = 0 \]
\[ F_1 = 1 \]
\[ F_n = F_{n-1} + F_{n-2}, \text{ with } n \geq 2 \]

Notation \( fib(n) = F_n \)

Specification:
\{ n \geq 0 \} S \{ x = fib(n) \}

Program \( S \equiv \)
\[ x := 0; \]
\[ y := 1; \]
\[ \text{count} := n; \]
\[ \text{while} \ \text{count} > 0 \ \text{do} \]
\[ h := y; \]
\[ y := x + y; \]
\[ x := h; \]
\[ \text{count} := \text{count} - 1 \]
\[ \text{end while} \]

Partial and total correctness: Fibonacci’s suite

Which intermediate formulas do we need to establish?

1. \( \vdash \{ n \geq 0 \} x := 0; y := 0; \text{count} := n; \{ p \} \)
   The invariant \( p \) is true when the while is reached:

2. \( \vdash \{ p \land \text{count} > 0 \} h := y; y := x + y; x := h; \text{count} := \text{count} - 1; \{ p \} \)
   The loops preserves the invariant:

3. \( \vdash \{ p \land \lnot(\text{count} > 0) \} \rightarrow x = fib(n) \)
   At the exit of the while, the postcondition is verified;
Partial and total correctness: Fibonacci's suite

For the total correctness:

1. $\{ t = z \} h := y; y := x + y; x := h; count := count - 1; \{ t < z \}$
   The termination function $t$ decreases at each loop's iteration.

2. $p \Rightarrow t \geq 0$
   The termination function has always a positive value when the invariant is true.

Which invariant do we need?

\[ p \equiv x = \text{fib}(n - count) \]
\[ \land y = \text{fib}(n - count + 1) \]
\[ \land count \geq 0 \]

Which termination function do we need?

\[ t \equiv count \]
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?

### Recalls

- **Semantic definition** of partial correctness:
  \[
  \models \{p\} S\{q\} \quad \text{iff} \quad \forall \sigma \in [p] : M[s](\sigma) = \emptyset \lor M[s](\sigma) \in [q] \quad \text{iff} \quad M[s](\{p\}) \subseteq [q]
  \]

- **Syntactical definition** of partial correctness:
  \[
  \vdash \{p\} S\{q\} \quad \text{iff} \quad \text{there exists a theorem in the proof system (corresponding to the partial correctness semantics) for the correctness formula.}
  \]
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 $\vdash_{\text{Tot}} \{p\} S\{q\}$
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)

\[ \text{wlp}(S, \Phi) / \text{wp}(S, \Phi) \]

- \( \text{wlp}(S, \Phi) \) is the set of proper states from which if \( S \) is executed, and if the execution terminates then the final state belongs to \( \Phi \);
- \( \text{wp}(S, \Phi) \) is the set of proper states from which if \( S \) is executed, the execution does terminate and the final state belongs to \( \Phi \).

Lemma

For any program \( S \) and assertion \( q \),

- there exists an assertion \( p \) such that \[ [p] = \text{wlp}(S, [q]) \]
- there exists an assertion \( p \) such that \[ [p] = \text{wp}(S, [q]) \]
Properties of the operators \( \mathsf{wlp} \) and \( \mathsf{wp} \)

For any program \( S, S_1, S_2 \) and assertions \( p \) and \( q \) :

1. \( \models \mathsf{wlp}(\mathsf{skip}, q) \iff q \)
2. \( \models \mathsf{wlp}(u := t, q) \iff q \mathrel{\triangleq} u := t \)
3. \( \models \mathsf{wlp}(S_1; S_2, q) \iff \mathsf{wlp}(S_1, \mathsf{wlp}(S_2, q)) \)
4. \( \models \mathsf{wlp}(\mathsf{if} \ B \ \mathsf{then} \ S_1 \ \mathsf{else} \ S_2, q) \iff (\mathsf{wlp}(S_1, q) \land B) \lor (\mathsf{wlp}(S_2, q) \land \neg B) \)
5. \( \models \mathsf{wlp}(S, q) \land B \iff \mathsf{wlp}(S_1, \mathsf{wlp}(S_2, q)) \), where \( S \equiv \mathsf{while} \ B \ \mathsf{do} \ S_1 \ \mathsf{od} \)
6. \( \models \mathsf{wlp}(S, q) \land \neg B \iff q \) where \( S \equiv \mathsf{while} \ B \ \mathsf{do} \ S_1 \ \mathsf{od} \)
7. \( \models \{p\}S\{q\} \iff p \rightarrow \mathsf{wlp}(S, q) \)

These properties are also true with the operator \( \mathsf{wp} \).

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 \( F : \Sigma \rightarrow \text{integer} \), there is an integer expression \( t \) such that for any proper state \( \sigma \), 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


