ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

# Testing Object-Oriented Software

**Didier Buchs, Cécile Péraire**

Department of Computer Science
Software Engineering Laboratory
September 1999



# Contents


- Introduction to test
- Functional Testing
- Functional Testing for Object-Oriented Software
- Operational Techniques and Test Set Generation Tool: *CO-OPNTEST*
- Case Study: Production Cell
- Conclusion
- References




## Introduction: Purpose of test

**Definition (IEEE 729):**

Testing is the process of exercising or evaluating a system or system component by manual or automated means to verify that it satisfies specified requirements or to identify differences between expected and actual results.

➥ Testing is the process of **finding** programs errors.

➥ Testing is not the process of **diagnosing** the cause of errors.

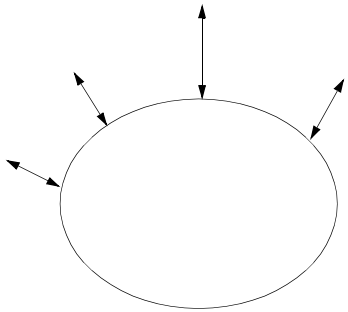
➥ Testing is not the process of **correcting** errors.

➥ Testing is not the process of **proving** the correction of the programs.



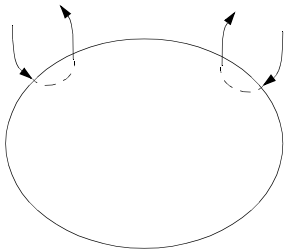
## Introduction: Kind of systems

**Client-Server Systems**



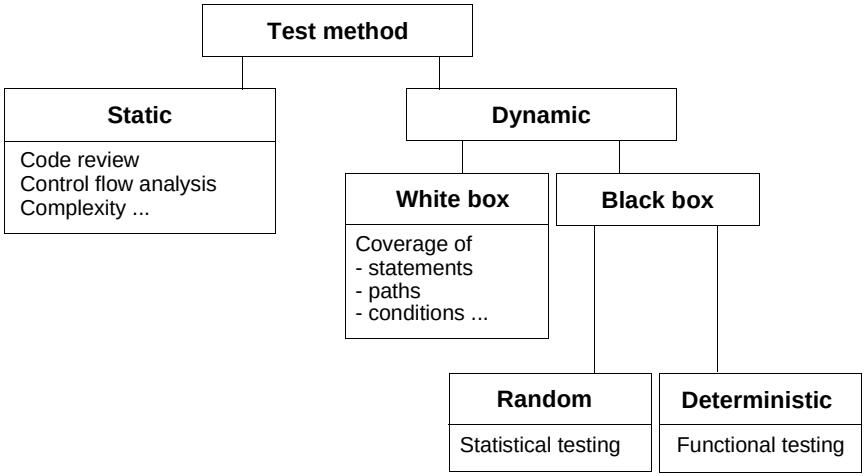
**Atomicity**
**Concurrency**
**Determinism or non-determinism**

**Reactive Systems**



**Concurrency**
**Non-determinism**

## Introduction: Classification of the test methods





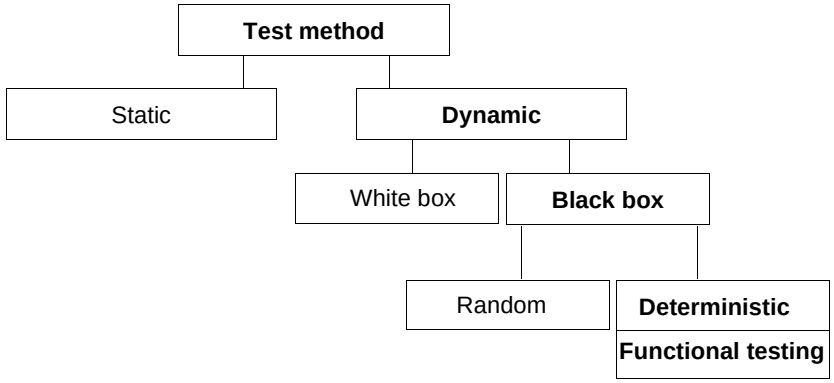

## Functional Testing

### Contents

- Purpose of functional testing
- Test Components
- Test process
- Test set selection
  - Uniformity hypotheses
  - Regularity hypotheses
  - Uniformity hypotheses with subdomains decomposition



## Purpose of functional testing




Functional testing is the process of **finding** programs errors by **running** the program under test with test sets derived from the **specification** of the application: for each **functionality** specified, a test set is selected.



## Test Components

T : test set in a language appropriate for exercising programs

**Term equality to apply on programs, sequence of function calls, ...**

SP : specification of the expected properties

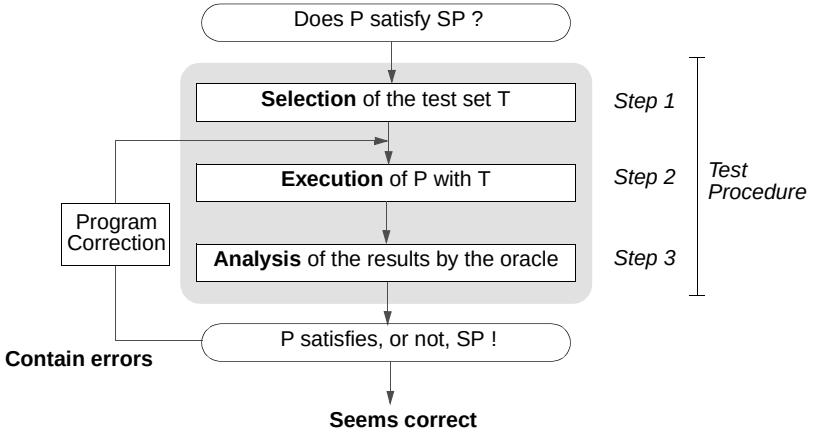
**Algebraic specification, Z, Petri nets, ...**

P : code of programs under test

**Ada, Java, ...**

## Test process

**Goal**: finding programs errors with respect to its specification.






## Test process

How to find a test set that can concretly be applied to programs in order to detect faults:

**Solution:**

Mimic the human intuition by the use of **hypothesis**

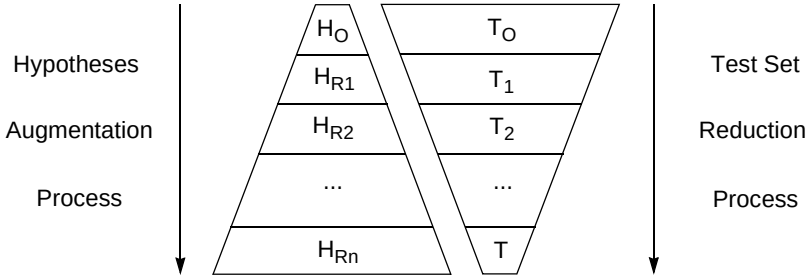
- on the structure of the execution sequence
- on the choice of values

Apply iteratively these hypotheses in order to **select the test exercising the program** on the possible errors



## Test process (cont'd)

**Goal of the reduction hypotheses $H_R$:** to reduce the size of the test set.




**Validity**: T rejects any incorrect program

**Unbiasedness**: T accepts any correct program



## Test process (cont'd)

Soient: $SPEC$ : class of all specifications,
$PROG$ : class of all programs,
$TEST$ : class of all test sets,
$\models$ : satisfaction relationship on $PROG \times SPEC$,
$\models_O$ : satisfaction relationship on $PROG \times TEST$.

***Step 1*** Selection of a test set T from a specification SP of the application and from a set of hypotheses H on the program under test.
$T \in TEST$, $SP \in SPEC$, $P \in PROG$, **(P satisfies H)** $\Rightarrow$ **$(P \models_O T \Leftrightarrow P \models SP)$**

***Step 2*** Execution of the program under test P using the test set T.

***Step 3*** Analysis of the results obtained during the execution of the program P.

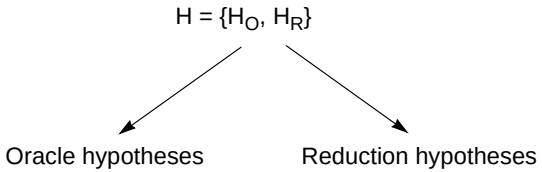
If the test set is successful ($P \models_O T$), then the test procedure is finished.

In addition, if P verifies the hypotheses H, then the program satisfies the requirements of the specification ($P \models SP$).

If the test set is not successful, then the program contains faults, and it is possible to return to the second step of the test procedure after correction.

## Test process (cont'd)

Notes concerning the hypotheses H:

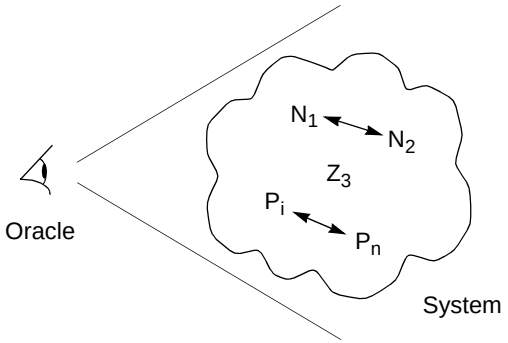





## Test process (cont'd)

**Goal of the oracle hypotheses $H_O$:** to guaranty the observability of the system.

➥ The oracle must be able to compare elements of same nature.






## Test set selection

In order to reduce the number of test cases, we perform reduction hypotheses $H_R$ on the behavior of the tested program.

| **Reduction hypotheses** |
| --- |
| • Uniformity hypotheses<br>• Regularity hypotheses |

Reduction hypotheses act on test cases with variables, replacing those variables so as to reduce the level of abstraction of the test formulas.



## Test set selection (cont'd)

Reduction hypotheses

Uniformity and regularity hypotheses can be combined in order to obtain more complex reduction hypotheses.

Since uniformity hypotheses are very strong, they are usually not applied to the elements *under test*, but to the elements *imported* into the specification, which we suppose being already tested. On the elements *under test*, we apply regularity hypotheses or uniformity hypotheses combined with subdomains decomposition.

# Functional Testing for Object-Oriented Software

## Contents

- Advantages and drawbacks of testing object-oriented software
- Choice of the specification and test formalisms
- *CO-OPN*
- Hennessy-Milner Logic (*HML*)
- Test set generation
- Test set reduction
- Incrementallity hypothesis in case of behavioral subtyping
- Abstract view of test selection process



## Advantages and drawbacks of testing object-oriented software (compared to traditional software)

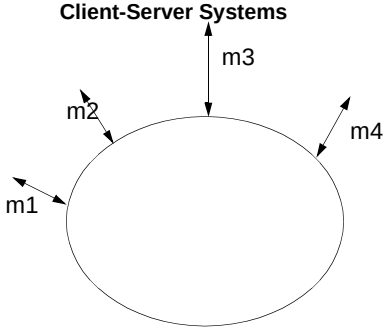
➠ The test unit is the **class** and not the function
(a “method” is tested via one instance of the class).

| ▲ Advantages | ▼ Drawbacks |
|---|---|
| • Encapsulation | • Encapsulated state<br>• Use of references to other objects |
| • Inheritance | • Polymorphism → dynamic binding (no static analysis) |
| • Behavioral subtyping (preservation of behaviors) | • Syntatic subtyping (no preservation of behaviors) |



## Kind of systems

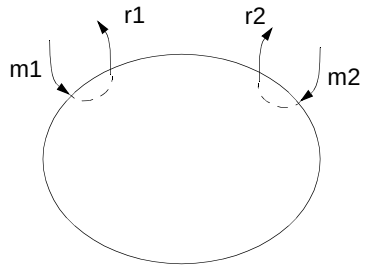
**Client-Server Systems**




**Atomicity**
**Concurrency**
**Determinism or non-determinism**

T= { <m1(in1,in2,..,out1,out2),
m2(in1,in2,..,out1,out2),
...>}

**Reactive Systems**




**Concurrency**
**Non-determinism**

T= { <m1(in1,in2,...),
{ r1(out1,out2), r2(out1,out2), ...}
m2(in1,in2,..,out1,out2),
{ r1(out1,out2), r2(out1,out2), ...}...>}



## Choice of the specification and test formalisms for client-server systems

- Formalism used to describe the **O-O specification** SP:
  *CO-OPN* [BBG97]
- Formalism used to describe the **test set** T:
  Hennessy-Milner Logic *HML* [HM85]

There is **full agreement** between
the equivalence relations of *CO-OPN* and *HML*

➥ (P satisfies H) ⇒ (P $\models_O$ T ⇔ P $\models$ SP)

# Specification Formalism

**CO-OPN: Concurrent Object-Oriented Petri Nets**

Specification language for complex distributed systems.

CO-OPN = Data Structures + Petri Nets + Object Orientation

- Data Structures = Algebraic Abstract Data Types
- Concurrency = Algebraic Petri Nets
- Modularity = Object orientation with strict encapsulation
- Dynamicity = References, Object Creation
- Communication = Synchronisation between Objects
- Distribution = Contexts = Units + Migration informations



**Properties:**

- Semantics of True Concurrency
- Equivalences, Refinements
- Verification by means of test selection techniques

**Structuring mechanisms**

- Genericity,inheritance, enrichment and subtyping
- Classes and instances
- Reference Algebras (OID)

**Development Tools (CoopnTool/SANDS))**

- Syntax Checker
- Simulation Tool
- Test set generators
- Graphical and textual viewer



**(CO-OPN/2)**

**Concurrent Object-Oriented Petri Nets**

| Components/Principles | Represented in CO-OPN/2 |
|---|---|
| Data structures | Algebraic specifications |
| Control structure, states, concurrency | Algebraic Petri nets |
| Objects, classes | Modular Algebraic Petri nets |
| Inheritance | Syntactic mechanisms |
| Sub-typing | Semantic constraint + order sorted algebra |
| Dynamic creation | Instantiation mechanism |



**Algebraic Abstract Data Types**

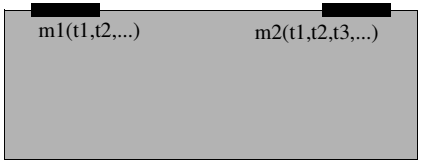
Abstract Data Type = Data structure(sort) + functions + properties

```
ADT Stack;
    INTERFACE
        USE Naturals, Booleans;
        SORTS stack;
    GENERATORS
        [] :-> stack;
        push _ _ :natural stack -> stack;
    OPERATIONS
        pop _    : stack      -> stack;
        top _    : stack      -> natural;
    BODY
    AXIOMS
        ax1: pop([])=[];
        ax2: pop(push(n,s))=s;
        ax3: top(push(n,s))=n;
    WHERE n : natural; s: stack;
END Stack;
```

## Algebraic Petri Nets: Objects

Object = State + Methods + Internal Transitions + Identity

- State = Petri Net Places
- Methods = parameterized transitions = Atomic Events
- Internal Transitions = Hidden Reactions
- Behavior = Petri Net Flow Relation



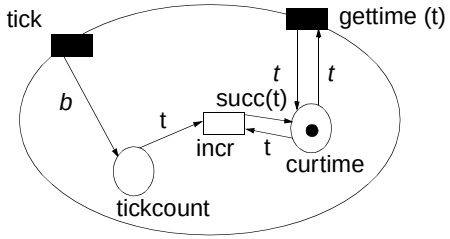

Object Interface with two methods



## Object: Timer

Object Timer that can be consulted (gettime) to give the current time, the event (tick) is used by a clock to increment the current time.

**Object** Timer;
**Interface**
**Use** Time, Booleans;
**Methods**
tick; gettime : time;
**Body**
**Places**
tickcount : boolean;
curtime : time;
**Initial**
curtime 0;
**Internal**
incr;
**Axioms**
axtick: tick :: -> tickcount(b);
axgettime: gettime(t):: curtime(t) -> curtime(t);
axincr: incr:: tickcount(b),curtime(t) -> curtime(succ(t));
**Where** b:boolean; t:time;
**End** Timer;






## Classes

A class is a template:

- Class type name
- Subtype relation
- Each instance has a reference which is a value of the class type (**`Type`**)
- Creation operation (**`Create`**).

Interaction of objects through synchronization operations

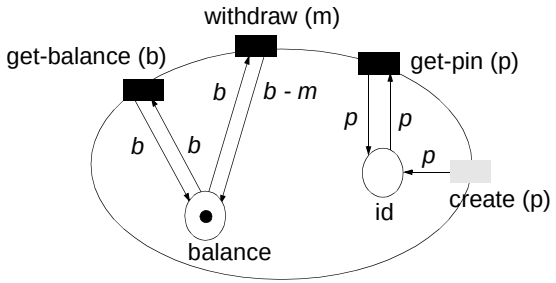
3 synchronization operators:

- & simultaneity
- .. sequence
- + alternative



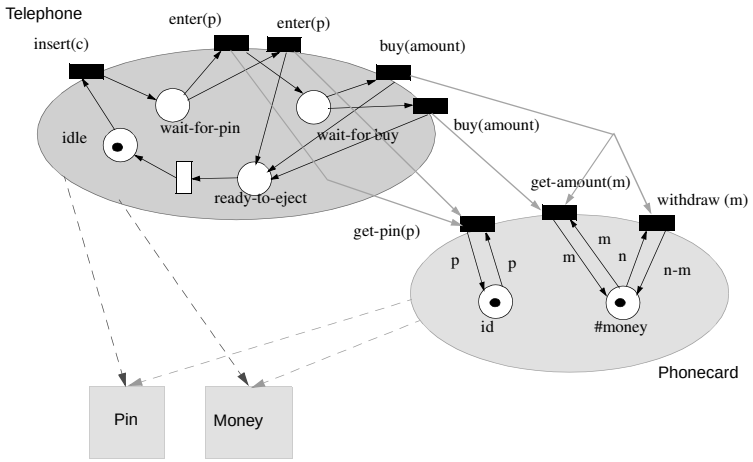
## Class: the phonecard

**Class** PhoneCard;
**Interface**
**Use** Booleans, Pin, Money;
**Type** phonecard;
**Creation** create _ : pin;
**Methods**
get-pin _ : pin;
get-balance _ : money;
withdraw _ : money;
**Body**
**Places**
balance: money;
id: pin;
**Initial**
balance 20;
**Axioms**
create p : -> id p;
get-pin p : id p -> id p;
get-balance b : balance b -> balance b;
b >= m => withdraw m : balance b -> balance b - m;
**Where** p : pin; b, m : money;
**End** PhoneCard;



## Incremental Specifications

Client class Telephone

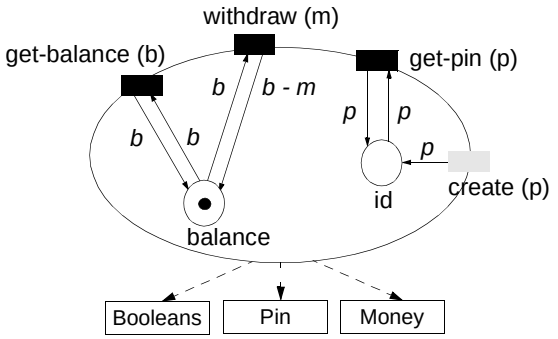





## *CO-OPN*

Example of a class: PhoneCard

**Class** PhoneCard;
  **Interface**
    **Use** Booleans, Pin, Money;
    **Type** phonecard;
    **Creation** create _ : pin;
    **Methods**
      get-pin _ : pin;
      get-balance _ : money;
      withdraw _ : money;
  **Body**
    **Places**
      balance: money;
      id: pin;
    **Initial**
      balance 20;
    **Axioms**
      create p : → id p;
      get-pin p : id p → id p;
      get-balance b : balance b → balance b;
      (b ≥ m) ⇒ withdraw m : balance b → balance b - m;
    **Where** p : pin; b, m : money;
**End** PhoneCard;






## Hennessy-Milner Logic (*HML*)

**Definition: Syntax of *HML***

For a set of events *E*, the *HML* language is defined as follows:

- $T \in HML$
- $f \in HML \Rightarrow (\neg f) \in HML$
- $f, g \in HML \Rightarrow (f \wedge g) \in HML$
- $f \in HML \Rightarrow (\langle e \rangle f) \in HML$ where $e \in E$

**Examples of *HML* formulas:**

$\langle e1 \rangle \langle e2 \rangle T$
$\langle e1 \rangle \langle e3 \rangle T$
$(\langle e1 \rangle \langle e2 \rangle T) \wedge (\langle e1 \rangle \langle e3 \rangle T)$
$\neg \langle e1 \rangle \langle e3 \rangle T$
...



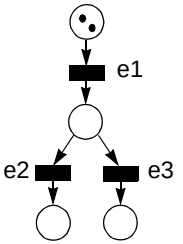
## Test set generation

A test is defined as a couple <*Formula, Result*> with:

- *Formula* ∈ *HML*
- *Result* ∈ {*true, false*}
  *Result* shows whether the expected result of the evaluation of *Formula* (from a given initial state) is *true* or *false*.

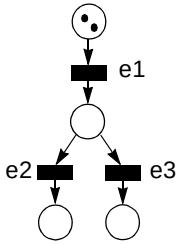
**Examples of tests:**

<<e1> <e2> *T, true*>
<<e1> <e3> *T, true*>
<(<e1> <e2> *T* ) ∧ (<e1> <e3> *T* ), *true*>
<<e1> <e2> <e3> *T*, *false*>
<¬ <e1> <e3> *T*, *false*>

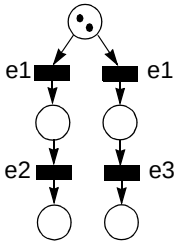



**Note**:
A test case allows verifying that:
- an acceptable behavior can be performed by the program,
- a non-acceptable behavior cannot be performed by the program.

## Test set generation - Advantage of *HML*




< <e1> ( <e2> *T* ∧ <e3> *T* ), *true* >

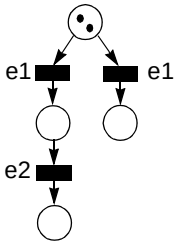



< <e1> ( <e2> *T* ∧ <e3> *T* ), *false* >

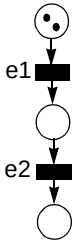
Hidden modification of system state



## Test set generation - Advantage of *HML* (cont'd)




< <e1> ¬ <e2> *T*, *true* >




< <e1> ¬ <e2> *T*, *false* >

Hidden modification of system state



## Test set reduction

Regularity hypothesis on the events of the *HML* formulas:
*"nb-events"*

**Hypothesis:**

If a test < *f*, *r* > is successful for all instances of *f*
having a number of events equal to a bound *k*,
then it is successful for all possible instances of *f*.

**Constraint:**

*nb-events* ( *f* ) = *k*

**Example:** *nb-events* ( *f* ) = *2*

`T1:<<e0><e1>T,result>`

`T2:<(<e0>T)and(<e1>T),result>`

`T3:<(not<e0>T)and(<e1>T),result>`

`T4:<(not<e0>T)and(not<e1>T),result>`



## Test set reduction (cont'd)

Regularity hypothesis on the events of the *HML* formulas:
*"nb-occurrences"*

**Hypothesis:**

If a test < *f*, *r* > is successful for all instances of *f*
having a number of occurrences of a given method *m* equal to a bound *k*,
then it is successful for all possible instances of *f*.

**Constraint:**

*nb-occurrences* ( *f*, *m*) = *k*

**Example:** *nb-occurrences* ( *f*, *withdraw*) = *2*

`T:<<create v0><withdraw v1><withdraw v2>T,result>`

## Test set reduction (cont'd)

Regularity hypothesis on the structure of the *HML* formulas:
*"shape"*

**Hypothesis:**

If a test < $f$, $r$ > is successful for all instances of $f$
having a given shape $s$,
then it is successful for all possible instances of $f$.

**Constraint:**

*shape* ( $f$, $s$ ) = *true*

**Example:** *shape* ( $f$, <create(123)> ( $f_0 \wedge \neg$ <get-balance(25)> $f_1$ )) = *true*

`T:<<create(123)>(`$f_0$ `and not <get-balance(25)>` $f_1$`),result>`



## Test set reduction (cont'd)

Uniformity hypothesis on the variables of the *HML* formulas:
*"uniformity"*

<<V1> <create V2> <withdraw V3> V4, Result>

- Uniformity on events (V1)
- Uniformity on algebraic values (V2, V3)
  with possible subdomains decomposition ("*subdomains-uniformity*")
- Uniformity on *HML* formulas (V4)

➽ The instantiation of the variables can also be done in the exhaustive way (*"exhaustiveness"*).



## Test set reduction (cont'd)

*Where to find subdomains?*

Subdomains are drawn up from the behavioral axioms:

Axiom$_m$: Condition $\Rightarrow$ Event ({ Parameter$_n$ }) **With** SynchroExpression:
Precondition $\rightarrow$ Postcondition

➽ In case of distinct axioms for one event,
subdomains of each axiom can possibly be overlapping.



## Test set reduction (cont'd)

*How to find subdomains in algebraic conditions?*

Each algebraic condition on an axiom will usually draw up 2 different subdomains, one for each possible result of the condition.

(b ≥ m) $\Rightarrow$ withdraw m : balance b $\rightarrow$ balance b - m

Two subdomains: (b ≥ m) = true
(b ≥ m) = false

*How to find subdomains in method parameters?*

get-balance b : balance b $\rightarrow$ balance b
$\Leftrightarrow$
(v = b) $\Rightarrow$ get-balance v : balance b $\rightarrow$ balance b

Two subdomains: (v = b) = true
(v = b) = false

## Test set reduction (cont'd)

*How to find subdomains in pre- and post-conditions (without synchronization expressions) ?*

$$\textit{CLASS}\ \frac{e :: pre_e \rightarrow post_e}{pre_e \xrightarrow{e} post_e} \qquad \textit{MONO}\ \frac{pre_e \xrightarrow{e} post_e}{s \xrightarrow{e} s'}$$

Three subdomains:

- $CS_1 = \{ (pre_e \subseteq s) = true, (s' = s - pre_e + post_e) = true \}$,
- $CS_2 = \{ (pre_e \subseteq s) = true, (s' = s - pre_e + post_e) = false \}$,
- $CS_3 = \{ (pre_e \subseteq s) = false \}$.



## Test set reduction (cont'd)

*How to find subdomains in pre- and post-conditions in the presence of synchronization expressions ?*

$$\textit{CLASS}\ \frac{e\ \textbf{With}\ synchro :: pre_e \rightarrow post_e}{} \qquad \textit{SYNC}\ \frac{e\ \textbf{With}\ synchro \quad pre_e \xrightarrow{} post_e \qquad s'' \xrightarrow{synchro} s'''}{s \xrightarrow{e} s'}$$

Four subdomains:

- $CS_1 = \{ \textit{Valid}\ \langle s'', synchro, s''' \rangle = true, \quad (pre_e \subseteq s) = true, \quad (s' = s''' + post_e) = true \}$,
- $CS_2 = \{ \textit{Valid}\ \langle s'', synchro, s''' \rangle = true, \quad (pre_e \subseteq s) = true, \quad (s' = s''' + post_e) = false \}$,
- $CS_3 = \{ \textit{Valid}\ \langle s'', synchro, s''' \rangle = true, \quad (pre_e \subseteq s) = false \}$,
- $CS_4 = \{ \textit{Valid}\ \langle s'', synchro, s''' \rangle = false \}$,

with $s'' = s - pre_e$.



## Test set reduction (cont'd)

*How to find subdomains in pre- and post-conditions in the presence of synchronization expressions ? -Single synchronization-*

$$\textit{CLASS}\ \frac{synchro = sync :: pre_{sync} \rightarrow post_{sync}}{} \qquad \textit{MONO}\ \frac{pre_{sync} \xrightarrow{sync} post_{sync}}{s'' \xrightarrow{sync} s'''}$$

$\textit{Valid}\ \langle s'', synchro, s''' \rangle = \textit{Valid}\ \langle s'', sync, s''' \rangle$



## Test set reduction (cont'd)

*How to find subdomains in pre- and post-conditions in the presence of synchronization expressions ? -Sequential synchronization-*

$$\textit{SEQ}\ \frac{s_{11} \xrightarrow{sync_1} s_{12} \qquad s_{21} \xrightarrow{sync_2} s_{22}}{s'' \xrightarrow{synchro = sync_1 .. sync_2} s'''}$$

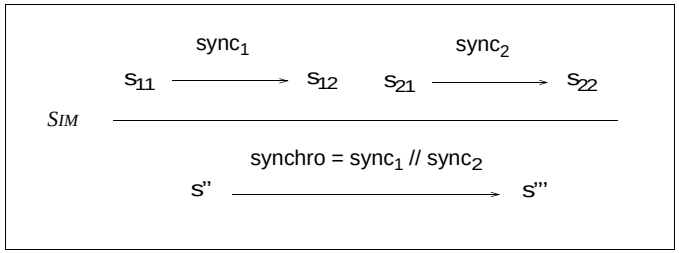
$\textit{Valid}\ \langle s'', synchro, s''' \rangle =$

$\textit{Valid}\ \langle s_{11}, sync_1, s_{12} \rangle \wedge \textit{Valid}\ \langle s_{21}, sync_2, s_{22} \rangle$

with $s'' = s_{11}$, $s_{12} = s_{21}$, $s''' = s_{22}$.

## Test set reduction (cont'd)

*How to find subdomains in pre- and post-conditions in the presence of synchronization expressions ?*
*-Simultaneous synchronization-*




*Valid* ⟨s'', synchro, s'''⟩ =

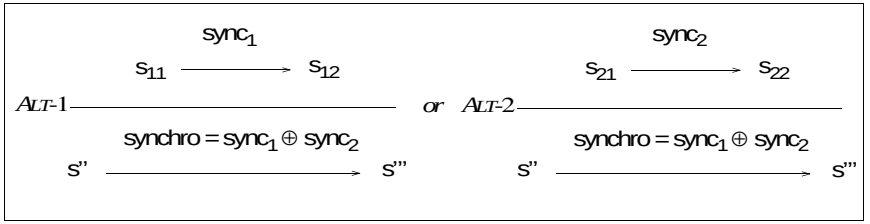
*Valid* ⟨$s_{11}$, $sync_1$, $s_{12}$⟩ ∧ *Valid* ⟨$s_{21}$, $sync_2$, $s_{22}$⟩

with (s'' = $s_{11}$ + $s_{21}$) and (s''' = $s_{12}$ + $s_{22}$).



## Test set reduction (cont'd)

*How to find subdomains in pre- and post-conditions in the presence of synchronization expressions ?*
*-Alternative synchronization-*




*Valid* ⟨s'', synchro, s'''⟩ = *Valid* ⟨$s_{11}$, $sync_1$, $s_{12}$⟩

with s'' = $s_{11}$ and s''' = $s_{12}$

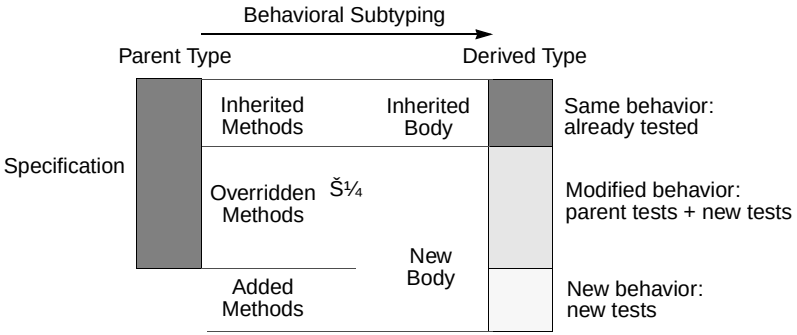
*or*

*Valid* ⟨s'', synchro, s'''⟩ = *Valid* ⟨$s_{21}$, $sync_2$, $s_{22}$⟩

with s'' = $s_{21}$ and s''' = $s_{22}$.



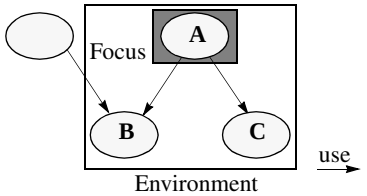
## Incrementallity hypothesis in case of behavioral subtyping

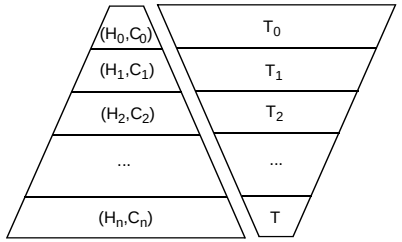





## Abstract view of test selection process

- Define a test environment
- Focus on a particular unit of interest
- Define a system of constraints on $T_0$:
  - Unit of interest: regularity hypotheses, uniformity hypotheses with subdomains decomposition
  - Other units: uniformity hypotheses
- Resolution of the system of constraints



# Operational Techniques and Test Set Generation Tool: *CO-OPNTEST*

## Contents

- Operational techniques (goal, need, means)
- Partial view of the *CO-OPNTEST* architecture
- Algebraic specifications of *CO-OPN*
  of *HML*
  of the *Constraints*
- Resolution techniques
- Subdomains decomposition (unfolding)
- *CO-OPNTEST*: Architecture
  Functionalities (from the user point of view)
  Graphical interface

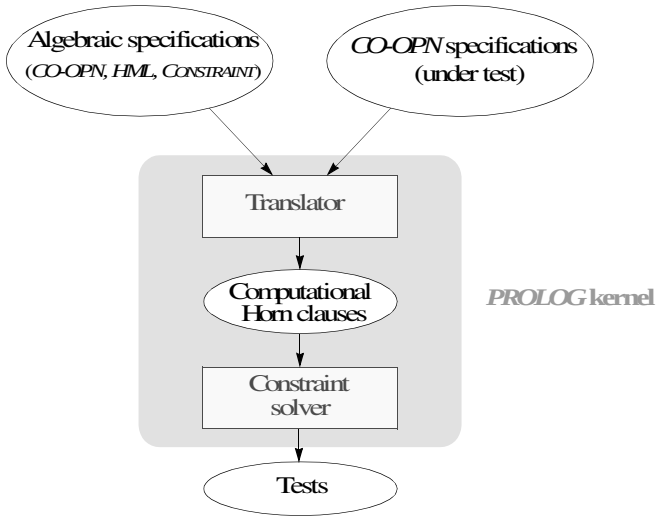


## Operational techniques

- **Goal**: Resolution of a constraints system.
- **Need**: Correct and complete **equational resolution procedure** (ex: “narrowing”).
- **Means**: **Simulation** of the equational resolution procedure by the ***PROLOG* SLD-resolution** associating a **Horn clause** to each constraint or axiom.



## Partial view of the *CO-OPNTEST* architecture






## Algebraic specifications of *CO-OPN*

Positive and Negative Semantics of *CO-OPN*

**Operation**:

```
valid _ _ _ : state synchro state -> boolean;
```

- **Positive semantics:**

$$state_1 \xrightarrow{sync} state_2$$

$$valid(state_1, sync, state_2) = true;$$

- **Negative semantics**:

$$state_1 \xrightarrow{sync} \not\quad state_2$$

$$valid\ (state_1, sync, state_2) = false;$$

## Algebraic specifications of *CO-OPN* (Cont'd)

**Adt** *CO-OPN*;
**Interface**
**Use** *Booleans, States, Events, Synchro*;
**Operations**
behavioralAxiom _ # _ ~> _ : event state state → boolean;
behavioralAxiom _ withsync _ # _ ~> _ : event synchro state state → boolean;
valid _ _ _ : state synchro state → boolean;

**Body**
**Axioms**

*;; Basic behavior*

(behavioralAxiom e # pre ~> post) =
cond & (pre ⊆ s) = bool1 &
(s' = s - pre + post) = bool2
⇒
valid (s, SYNCHRO e, s') =
cond and (bool1 and bool2);

*;; Basic behavior with synchronization*

(behavioralAxiom e With sync # pre ~> post) =
cond & (pre ⊆ s) = bool1 &
(s' = s''' + post) = bool2 &
valid (s - pre, sync, s''') = bool3 ⇒
valid (s, SYNCHRO e, s') =
cond and (bool1 and bool2) and bool3;



## Algebraic specifications of *CO-OPN* (Cont'd)

;; *Sequential synchronization*
valid ( s11, sync1, s12 ) = bool1 &
valid ( s12, sync2, s22 ) = bool2 ⇒
valid ( s11, sync1 .. sync2, s22 )
= bool1 and bool2;

;; *Simultaneous synchronization*
valid ( s11, sync1, s12 ) = bool1 &
valid ( s21, sync2, s22 ) = bool2 ⇒
valid ( s11 + s21, sync1 // sync2, s12 + s22 )
= bool1 and bool2;

;; *Alternative synchronization*
valid ( s11, sync1, s12 ) = bool1 ⇒
valid ( s11, sync1 ⊕ sync2, s12 ) = bool1;

valid ( s21, sync2, s22 ) = bool2 ⇒
valid ( s21, sync1 ⊕ sync2, s22 ) = bool2;

**Where**
e : event;
sync, sync1, sync2 : synchro;
s, s', s'', s''', s11, s12, s21, s22 : state;
cond, bool1, bool2, bool3: boolean;
**End** *CO-OPN*;



## Algebraic specifications of *HML*

**Adt** *HML*;
**Interface**
**Use** *CO-OPN*;
**Sort** hml;
**Generators**
*T* : → hml;
< _ > _ : event hml → hml;
_ and _ : hml hml → hml;
not _ : hml → hml;
**Operation**
*;; satisfaction:*
*;; s ⊨ f : s satisfies the HML formula f*
_ ⊨ _ : state hml → boolean;

**Body**
**Axioms**
s ⊨ *T* = true;

valid (s, SYNCHRO e, s') = true ⇒
s ⊨ <e> f = s' ⊨ f ;

valid (s, SYNCHRO e, s') = false ⇒
s ⊨ <e> f = false;

(s ⊨ (f and g)) = ((s ⊨ f) and (s ⊨ g));

(s ⊨ not f) = not (s ⊨ f );

**Where**
f, g : hml; e : event; s, s' : state;
bool, bool1, bool2 : boolean;
**End** *HML*;



## Algebraic specifications of the constraints

**Adt** *CONSTRAINT*;
**Interface**
**Use** *Naturals*, *HML*;
**Operations**
nb-events _ : hml → natural;
shape _ _ : hml hml → boolean;
...
validation _ : hml → boolean;
**Body**
**Axioms**
*;; Number of events of the HML formula*
nb-events *T* = 0;
nb-events <e> f = succ ( nb-events f );
nb-events ( f and g ) =
( nb-events f ) + ( nb-events g );
nb-events not f = nb-events f ;

*;; Shape of the HML formula:*
*;; shape (f, s): f must have the shape s*
shape *T T* = true;
shape (<e> f) (<e> s) = shape f s;
shape (f1 and g1) (f2 and g2) =
(shape f1 f2) and (shape g1 g2);
shape (not f) (not s) = shape f s;
...
*;; validation from the initial state* initstate
validation f = (initstate ⊨ f);
**Where**
f, g, f1, f2, g1, g2, s: hml;
e, e1, e2: event;
st : states;
**End** *CONSTRAINT*;

## Resolution techniques

**Goal**: **Simulation** of the equational resolution procedure
by the ***PROLOG* SLD-resolution**
associating a **Horn clause** to each constraint or axiom:

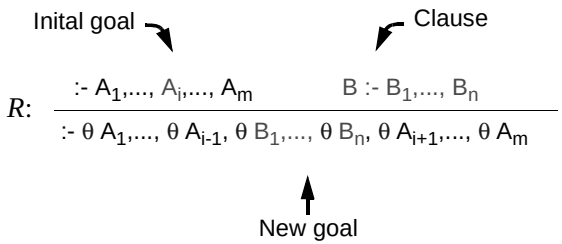
Axioms: addition (0, n) = n
addition (succ (n), m) = succ (addition (n, m))

Horn clauses: addition (0, n, n)
addition (succ (n), m, succ (r)) :- addition (n, m, r)



## Resolution techniques (Cont'd)

Standard *PROLOG* resolution rule




θ : *Most General Unifier* of $A_i$ and B



## Resolution techniques (Cont'd)

Standard *PROLOG* resolution strategy: SLD-resolution
(Linear resolution with Selection function for Definite clauses)

$$R: \frac{\text{Goal} \qquad \text{Clause}_1}{\text{Goal}_1} \text{ with } \theta_1$$

$$R: \frac{\text{Goal}_1 \qquad \text{Clause}_2}{\text{Goal}_2} \text{ with } \theta_2$$

$$\vdots$$

$$R: \frac{\text{Goal}_{n-1} \qquad \text{Clause}_n}{\text{Goal}_n} \text{ with } \theta_n$$

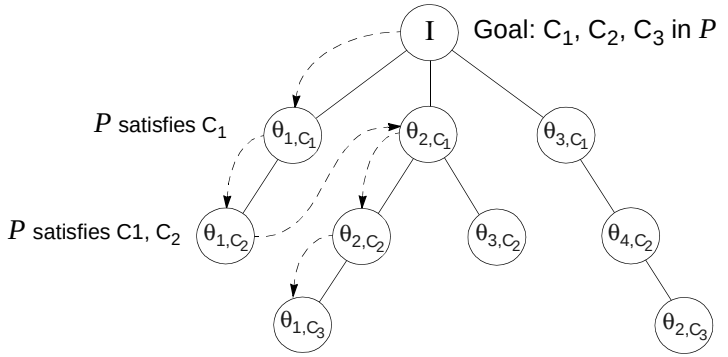
- A ***successful*** SLD-derivation ends with $\text{Goal}_n = \square$ (empty goal):
  an answer is the substitution $\theta_1.\theta_2...\theta_n$ (restricted to the variables of Goal).
- A ***failed*** SLD-derivation ends with $\text{Goal}_n \neq \square$ (non-empty goal).



## Resolution techniques (Cont'd)

Standard *PROLOG* resolution strategy: SLD-resolution

**Computation rule**: Selection of the left-most atom $C_1$ of **Goal: $C_1$,..., $C_i$,..., $C_m$**
**Search rule**: Depth-first-search

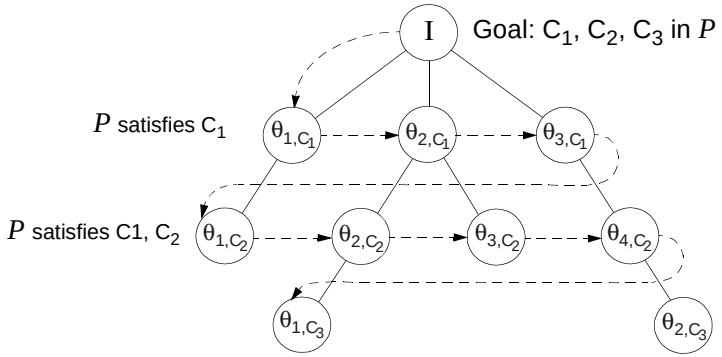



▲ **Correct** and **efficient** strategy
▼ **Uncomplete** strategy (**unfair** search rule):
no guarantee to find all the success branches in case of infinite branches

## Resolution techniques (Cont'd)

Complete SLD-resolution

**Computation rule**: Selection of the left-most atom $C_1$ of **Goal: $C_1$,..., $C_i$,..., $C_m$**
**Search rule**: Breadth-first-search




▲ **Correct** and **complete** strategy (**fair** search rule)
▼ **Non efficient** strategy



## Resolution techniques (Cont'd)

*CO-OPNTEST* SLD-resolution

**Computation rule**: Selection of the atom $C_i$ of **Goal: $C_1$,..., $C_i$,..., $C_m$**
*which **unifies** with a **minimum** of **clause heads**.*
➠ **Unsatisfaction detection mechanism**
(If $C_i$ unifies with no clause head, then **Goal** is unsatisfied).

**Search rule**: Depth-first-search

▲ **Correct** and **efficient** strategy.

▼ **Uncomplete** strategy (**unfair** search rule):
no guarantee to find all the success branches in case of infinite branches.

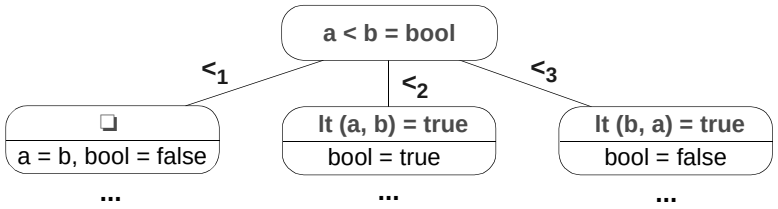
▲ **Resolution termination increased** by **unsatisfaction detection mechanism**.



## Subdomains decomposition (unfolding)

Unfolding = SLD-resolution + Control (wait)

| **< :** | **natural, natural -> boolean;** | **lt :** | **natural, natural -> boolean;** |
|---|---|---|---|
| $<_1$ : | x < x = false; | $lt_1$ : | lt (x, 0) = false; |
| $<_2$ : | lt (x, y) = true => x < y = true; | $lt_2$ : | lt (0, succ y) = true; |
| $<_3$ : | lt (y, x) = true => x < y = false; | $lt_3$ : | lt (succ x, succ y) = lt (x, y); |

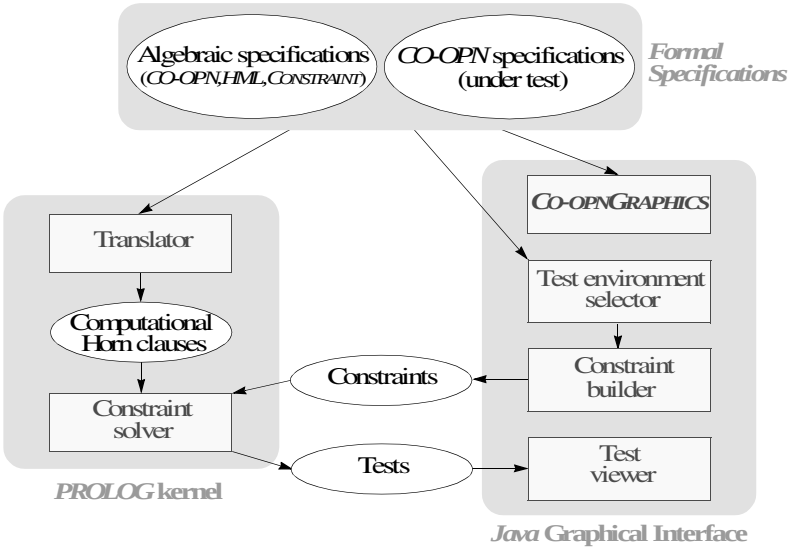



**Control procedure: wait ( lt (a, b, true ) ) :- var (a), var (b).**

➥ A solution of a goal is a couple (substitution, constraints)



## *CO-OPNTEST* architecture



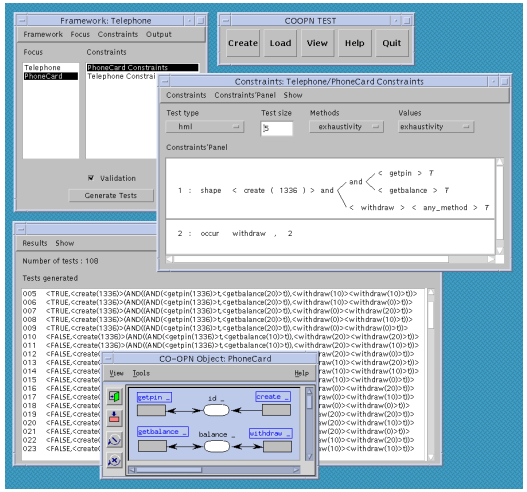
## *CO-OPNTEST* functionalities
(from the user point of view)

- Display of the specification graphs (via *CO-OPNGRAPHICS*)
- Definition of the test focus and environment
- Definition of constraints applied to the exhaustive test set:
  - **Regularity** applied to the formulas
  - **Exhaustiveness** applied to the variables
  - **Uniformity** applied to the variables
  - **Subdomains decomposition**
  - **Validation** of the tests <*HML Formula, Result*>
- Resolution of the system of constraints
- Display of the test set



## *CO-OPNTEST* graphical interface

Snapshot of the test selection for the class PhoneCard






# Case Study: Production Cell

### Contents

- Goal and Features
- Software Development Process
- Description of the Production Cell
- Unit Testing of a Device: the Press



## Goal and Features

**Goal:**

➤ Test of a **control program** for an existing industrial production cell.

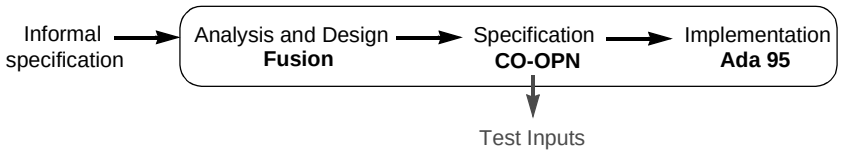
**Features:**

➤ **Realistic** industry-oriented problem,
taken from a real-world industrial plant in Karlsruhe.

➤ **Safety critical** system.

➤ **Reactive** system.

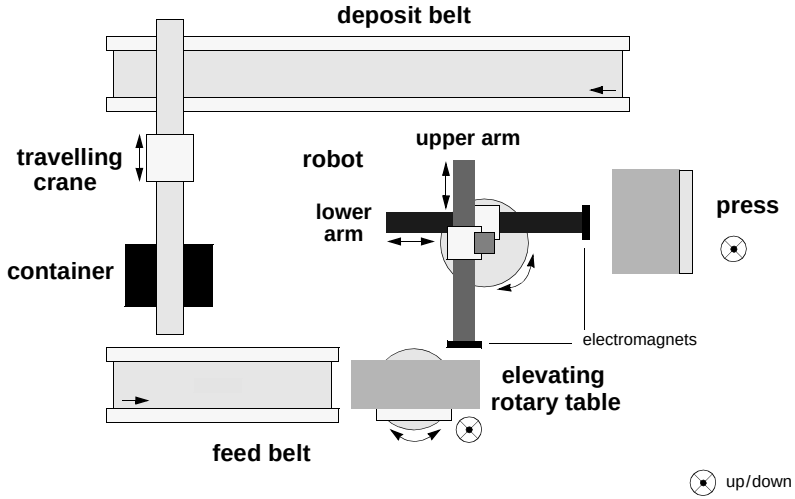
## Software Development Process




- Analysis and design: **Fusion** (18 classes).
- Formal specification: **CO-OPN** (1000 lines).
- Implementation: **Ada 95** (2000 lines).



## Description of the Production Cell (top view)

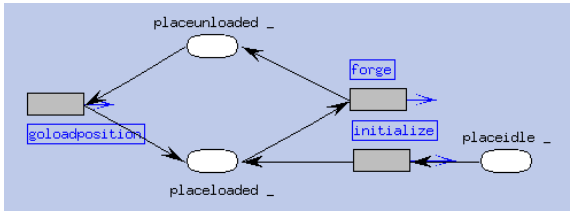





## Unit Testing of a Device: the Press

The Press device is tested using a stub that simulates the behavior of the Robot.

**Quality criteria**: coverage of the various specification cases (wrt the hypotheses).




Petri net of the Press



## Unit Testing of a Device: the Press (Cont'd)

Constraints

- **nb-events ( f ) = 5** -- *f* is a *HML* formula with 5 calls of method
- **trace ( f ) = true** -- *f* is a *HML* formula without *And* and *Not* operators
- **nb-occurrences ( f, 'initialize' ) = 1**
  -- f is a *HML* formula having one occurrence of initialize
- **shape (f, next ('initialize', [ ] )) = true**
  -- f is a *HML* formula having the shape: <initialize> [ ]

The variables (of type event) are replaced in an exhaustive way.

Those constraints produce the following tests:

## Unit Testing of a Device: the Press (Cont'd)

01: <FALSE,<initialize><goloadposition><goloadposition><goloadposition><goloadposition>T>
02: <FALSE,<initialize><goloadposition><goloadposition><goloadposition><forge>T>
03: <FALSE,<initialize><goloadposition><goloadposition><forge><goloadposition>T>
04: <FALSE,<initialize><goloadposition><goloadposition><forge><forge>T>
05: <FALSE,<initialize><goloadposition><forge><goloadposition><goloadposition>T>
06: <FALSE,<initialize><goloadposition><forge><goloadposition><forge>T>
07: <FALSE,<initialize><goloadposition><forge><forge><goloadposition>T>
08: <FALSE,<initialize><goloadposition><forge><forge><forge>T>
09: <FALSE,<initialize><forge><goloadposition><goloadposition><goloadposition>T>
10: <FALSE,<initialize><forge><goloadposition><goloadposition><forge>T>
11: **<TRUE,<initialize><forge><goloadposition><forge><goloadposition>T>**
12: <FALSE,<initialize><forge><goloadposition><forge><forge>T>
13:<FALSE,<initialize><forge><forge><goloadposition><goloadposition>T>
14: <FALSE,<initialize><forge><forge><goloadposition><forge>T>
15: <FALSE,<initialize><forge><forge><forge><goloadposition>T>
16: <FALSE,<initialize><forge><forge><forge><forge>T>

Assuming that the initialization mechanism is already tested,
this test set tests 2 cycles of the press use:
it insures that the first cycle of the press <forge><goloadposition>
is followed by another cycle <forge><goloadposition>.



## Conclusion

- Testing is the process of **finding** programs errors.
- An efficient program verification method is an adequate combination of use of different static and dynamic testing methods.
- The theory of functional testing can be adapted to object-oriented software development by using a new theoretical framework and the adequate reduction hypotheses.
- Possible automatization: a *Prolog* tool (*CO-OPNTEST*) has been implemented, by coding the *HML* semantics rules and the *CO-OPN* class semantics into equational logic: resolution techniques allow to compute tests sets from *CO-OPN* specifications, while a front-end written in Java allows a user-friendly definition of the constraints.
- Validation with a case study of industrial size: the Production Cell.



## References

[Pressman92] **Pressman R.**,
*Software Engineering - A Practitioner's approach*,
McGraw-Hill International Editions, 1992.

[Laprie95] **Laprie J.-C.**, *Guide de la sûreté de fonctionnement*,
Cépaduès-Editions, 1995.

[Myers79] **Myers G.**, *The Art of Software Testing*, Wiley, 1979.

[Beizer83] **Beizer B.**, *Software Testing Techniques*,
Van Nostrand Reinhold, 1983.



## References (Cont'd)

[BGM91] **Bernot G.**, **Gaudel M.-C.**, **Marre B.**,
"Software Testing Based on Formal Specification:a Theory and a Tool",
*Software Engineering Journal*, Vol 6, Num 6, nov. 1991, pp 387-405.

[Marre91] **Marre B.**, *Une méthode et un outil d'assistance à la sélection de jeux de tests à partir de spécifications algébriques*,
Thèse no 1561 - Université de Paris-Sud, 1991.

[BBG97] **Biberstein O.**, **Buchs D.**, **Guelfi N.**,
"CO-OPN/2: a concurrent object-oriented formalism",
In Gul Agha and Fiorella de Cindio, editors,
*Advances in Petri Nets on Object-Orientation*,
LNCS (Lecture Notes in Computer Science), Springer Verlag, 1999.

[HM85] **Hennessy M.**, **Milner R.**,
"Algebraic laws for nondeterminism and concurrency",
*Journal of the ACM*, 32(1):137-161, January 1985.

## References (Cont'd)


[PBB98] **Péraire C.**, **Barbey S.**, **Buchs D.**,
"Test Selection for Object-Oriented Software Based on Formal Specifications", *Proceedings of PROCOMET'98 (Programming Concepts and Methods)*, New York, USA, June 1998,
Also available as Technical Report (EPFL-DI No 97/252).

[BBP96] **Barbey S.**, **Buchs D.**, **Péraire C.**,
"A Theory of Specification-Based Testing for Object-Oriented Software",
*EDCC-2 (European Dependable Computing Conference)*,
LNCS (Lecture Notes in Computer Science) 1150, Springer verlag,
Taormina, Italy, October 1996, pp. 303-320,
Also available as Technical Report (EPFL-DI No 96/163).




## References (Cont'd)


[Per98] **Péraire C.**,
*Formal Testing of Object-Oriented Software: from the Method to the Tool*, Ph.D. Thesis, no 1904,
EPFL-DI, Switzerland , December 1998.

[Barb97] **Barbey S.**,
*Test Selection for Specification-Based Testing of Object-Oriented Software Based on Formal Specifications*, Ph.D. Thesis, no 1753,
EPFL-DI, Switzerland , December 1997.