Model Based Testing from Behavioural Models using Constraint Logic Programming

TAROT'2009 Summer School

14/07/09

Frédéric Dadeau

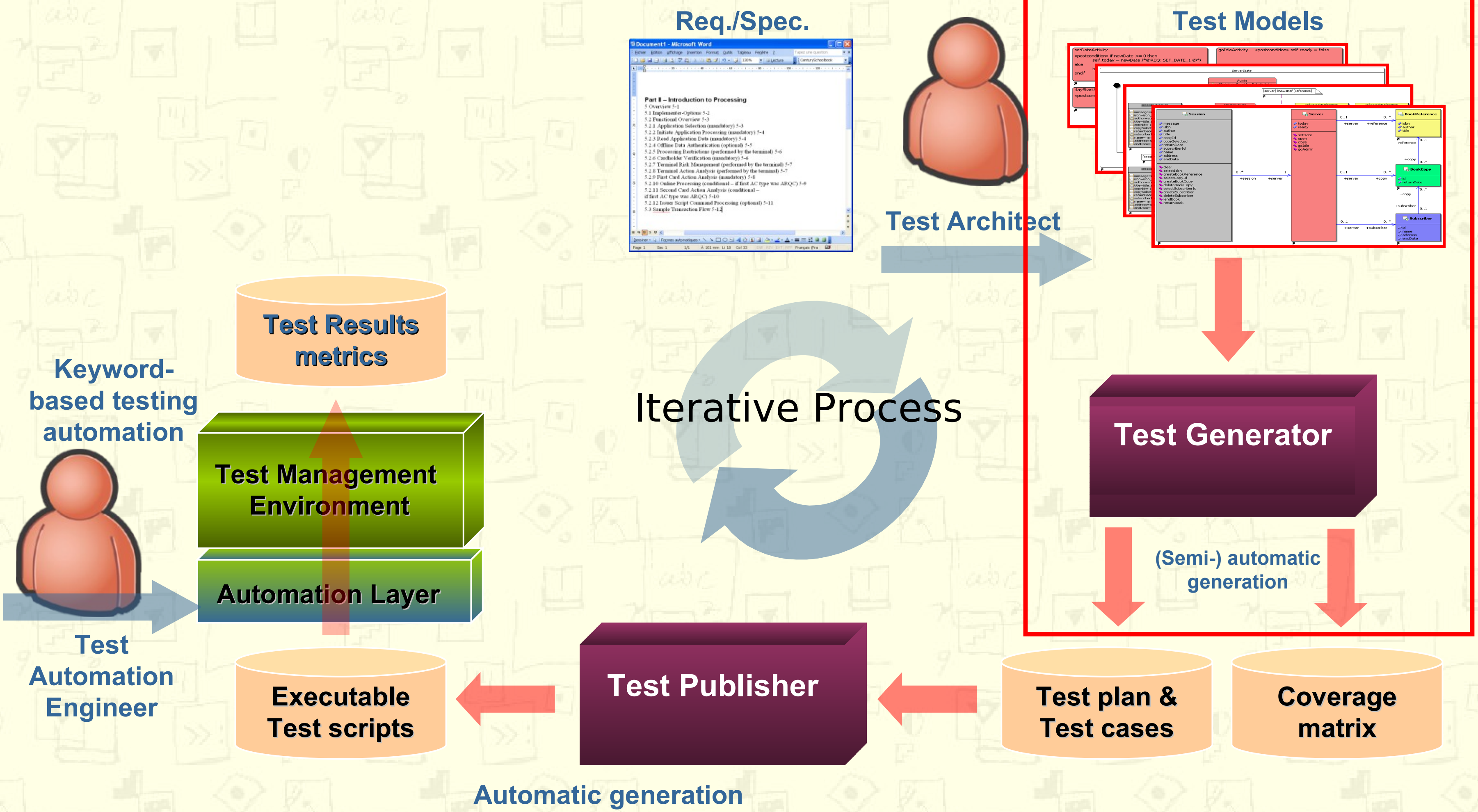
LIFC - INRIA CASSIS
Besançon, France

frederic.dadeau@lifc.univ-fcomte.fr

INRIA

LIFC

Model Based Testing
A general approach



Model-Based Testing
A few introductory points

- Models give an operational view of the system (behavioural models)
  - Operations describe transitions of the systems
  - No known topology of the system (possibly billions of states)
  - Work initially done on B machines, extended later with Statecharts, JML, UML/OCL
- **C**onstraint **L**ogic **P**rogramming has two main uses:
  - Computing the test targets (involving a boundary analysis)
  - Animating the model to reach the test targets

- An insight into a research work …
  - named the BZ-Testing-Tools project
  - done at the LIFC in Besançon since early 2000
  - in the activity group led by Pr. Bruno Legeard
  - involving numerous (and valuable!) Ph.D. students and researchers: Dr. Laurent Py, Pr. Fabrice Bouquet, Dr. Fabrice Ambert, Dr. Fabien Peureux, Dr. Franck Lebeau, Dr. Séverine Colin, Dr. Nicolas Vacelet, Dr. Frédéric Dadeau, Dr. Vincent Pretre, Régis Tissot, Pierre-Christophe Bué, Jonathan Lasalle, …
  - that led to the creation of a start-up company in 2003: Leirios Technologies renamed Smartesting in 2008: http://www.smartesting.com

Outline

1. Notions of Constraint Logic Programming
2. Symbolic animation of models
3. Automated boundary test generation
4. Industrial experience
5. Scenario-Based Testing
6. Conclusions and future work

1. Notions of Constraint Logic Programming
    1. Logic Programming
    2. Constraint Logic Programming
    3. Constraint solvers
    4. Consistency checking algorithms
    5. Problem resolution

- Broad definition:
  "use of mathematical logic for computer programming"

- How does it work?
  - Set of elementary facts
  - Logical rules having more or less direct consequences
  - A resolution engine exploits them to answer a query

- Famous logic programming engine: Prolog (1972-74)
  - Many implementations SICStus, GNU, ECLiPSe, SWI, …

```
man(anakin).
man(luke).
man(obiwan).
woman(leia).
woman(padme).
wookie(chewbacca).
parent(luke, anakin, padme).
parent(leia, anakin, padme).
```

Set of facts

Main notions:
- Unification
- Backtracking

Resolution of queries using a top-down approach

```
?- man(M).
M = anakin ;
M = luke ;
M = obiwan ;
no.
```

```
sibling(Person1,Person2) :-
     parent(Person1, Fa, Mo),
     parent(Person2, Fa, Mo),
     Person1 /= Person2.

sister(Person,Sister) :-
     sibling(Person,Sister),
     woman(Sister).

brother(Person,Brother) :-
     sibling(Person,Brother),
     man(Brother).
```

```
?- trace, sister(luke,S).
call: sibling(luke,S)
call: parent(luke, Fa, Mo)
exit: parent(luke, anakin, padme)
call: parent(Person2, anakin, padme)
exit: parent(luke, anakin, padme)
fail: luke /= luke
redo: parent(leia, anakin, padme)
exit: luke /= leia
exit: sibling(luke, leia)
exit: woman(leia)
exit: sister(luke,leia)
S = leia ;
no.
```

Difference with “simple” Logic Programming

```
a(X,Y) :- X > 0, b(X, Y).
b(X,1) :- X < 0.
b(X,Y) :- X = 1, Y > 0.
```

Query: a(X,1). What happens?
Execution raises an “instantiation error” on X

Using Constraint Logic Programming, this succeeds!
→ X > 0 is a constraint that will be kept in a “constraint store”

- Nevertheless, using constraints is not so simple:
  - Constraints are not naturally managed by Prolog
  - Dedicated constraint solvers have to be written to handle specific data types (e.g. lists, sets)
  - Fortunately, numerous libraries exist, e.g. CLP(FD) for finite domain integers

- What do we need to design our constraint solver?
  - A constraint language (operators)
  - A constraint representation system
  - A constraint consistency checking mechanism
  - A resolution procedure

A CSP is a quadruplet(X,D,C,R), where:

- $X = \{x_1, \ldots, x_n\}$ is a set of n variables
- $D = \{D_1, \ldots, D_n\}$ is a set of domains, with $x_i \in D_i$
- $C = \{C_1, \ldots, C_m\}$ is a set of m constraints
- $R = \{R_1, \ldots, R_m\}$ is a set of relations, associated to each constraint

Relations express the compatibility between the values of the variables involved in the constraint.

Relations can be:

- unary: on one single variable $X * X = 1, X \in [-10..10]$
- binary: two variables are related two at a time $X * Y > 2, X < Y$
- …
- n-ary: relation between all the variables all_different([X_1,X_2,…])

Properties on the constraints

- Satisfiable: there exists a solution to the constraint

  $X = 1, Y = X + 1$

- Unsatisfiable: there are no solutions to the constraint

  $X < 3, Y = X + 2, Y > 6$

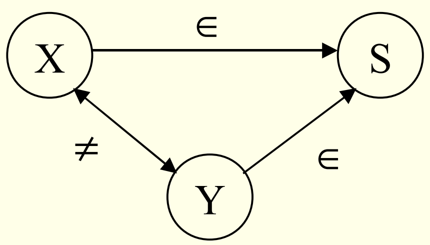
Constraints are stored in a “constraint store” that can be:

- Consistent: there exist a valuation of the variables that makes all the constraints of the store satisfiable
- Inconsistent: No valuation of the variables can make all the constraints satisfiable

The valuation procedure is called labeling.

A common representation is to use a graph:

- Nodes are variables and their current domains
- Arcs are constraints between the variables



All variables must have a domain!

D_S = {∅, {1}, {2}, {1,2}}

D_X = {1, 2}

D_Y = {1, 2}

Variable domains can be represented in several ways:

- Enumerated, e.g. {4,8,15,16,24,42}
- Intervals, e.g. [4..16] \ {8,15} or [4-8] ∪ {15,16} ∪ {24}
- A combination of both

Constraint graph consistency algorithm

- **Category:** algorithms on graphs
- **Goal:** eliminate inconsistent values from the graph
- **Methods :**
  - On enumerated domains: arc-consistency, path-consistency, k-consistency
  - On intervals: 2B-consistency, box-consistency

Partial methods: do not guarantee the existence of a solution

In the end, need to instantiate the variables to ensure consistency

Methods on enumerated domains

- Arc consistency: uses the arcs to reduce the domains of the variables
  - Each pair of variables is considered
  - Algorithms AC1 to AC7
    - AC1, AC2, AC3 - Mackworth 1977
    - AC4 - Mohr & Henderson 1986
    - AC5 - Van Hentenryck, Deville, Teng 1992
    - AC6 - Bessiere 1994
    - AC7 - Bessiere, Freuder, Régin 1999
- K-consistency: checks the consistency of one variable w.r.t. (k-1) other variables
  - Arc-consistency = 2-consistency
  - Path-consistency = 3-consistency

Some definitions related to arc-consistency

Let G = (X,D,C,R) be a constraint graph

- $R_{ij} \in C$ the relation between variable x_i and x_j

- x_i is arc-consistent w.r.t. x_j if and only if

  $\forall\, a_i \in D_i, \exists\, a_j \in D_j$ such that $(a_i, a_j) \in R_{ij}$

- A sub-graph (x_i, x_j) is arc-consistent iff
  - x_i is arc-consistent w.r.t. to x_j and
  - x_j is arc-consistent w.r.t. to x_i

- More generally, a constraint graph is arc-consistent iff
  - all its arcs are arc-consistent

Algorithm AC1

- Brute force method for checking the consistency of a graph

- Algorithm:

```
repeat
    reduce ← false
    for all (xi, xj) do
        reduce ← reduce ∨RE
        reduce ← reduce ∨RE
    done
while reduce
```

- complexity : $O(enk^3)$
  - e: number of constraints
  - n: number of variables
  - k: size of domains

REVISE ensures the arc-consistency of variable x_i w.r.t. variable x_j

```
boolean REVISE(xi, xj)
  delete ← false
  for each ai ∈ Di
    if ¬ (∃ aj ∈ Dj / (ai, aj) ∈ Rij)
      remove ai from Di
      delete ← true
    endif
  done
  return delete
```

Complexity : $O(k^2)$

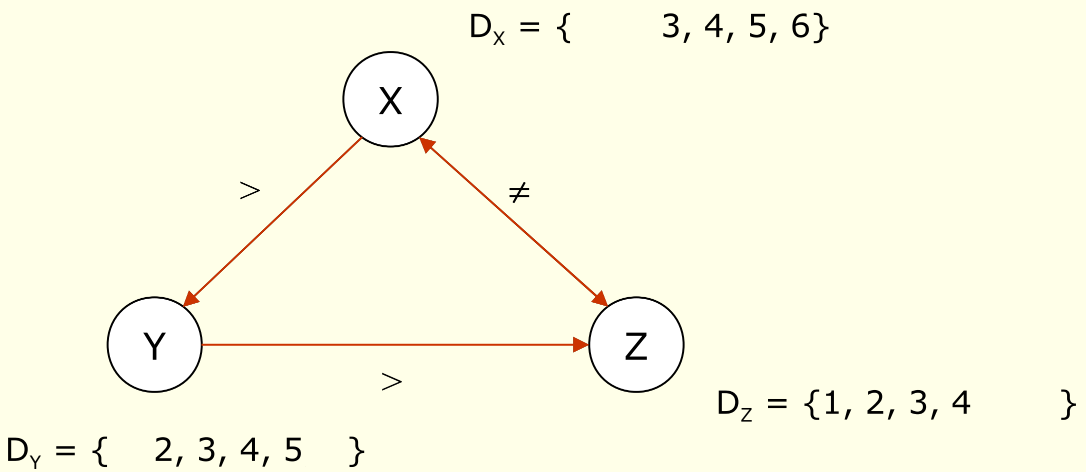
Algorithm AC3

- Algorithm using a queue of arcs to be re-examined:

```
queue ← ∅
for all (x_i, x_j) do
    queue ← queue ∪ {(x_i, x_j), (x_j, x_i)}
done
while (queue ≠ ∅) do
    (x_i,x_j) ← queue.pop()
    reduce ← REVISE(x_i,x_j)
    if (reduce)
        queue ← queue ∪ {(x_k, x_i) | k ≠ i ∧ k ≠ j }
    endif
done
```

- complexity : $O(ek^3)$
  - e: number of constraints
  - k: size of domains

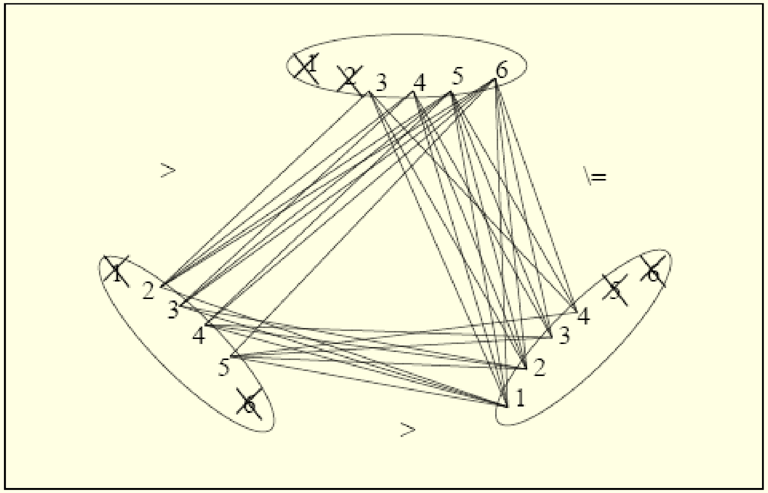
Example of AC3 algorithm: $X > Y$, $Y > Z$, $X \neq Z$



Queue =   (X,Y) (Y,X) (Y,Z) (Z,Y) (X,Z) (Z,X) (X,Y) (Z,X)

A word on AC4 algorithm

- Idea: relate variable values to each others for each constraint



- Improved complexity $O(ek^2)$ (vs. $O(ek^3)$ for AC3), but …
- Involves a matrix when implemented (lists of values in AC3)

Methods on intervals

- 2B-consistency, approximation of arc-consistency
  - A constraint *c* is 2B-consistent if
    - for each variable x of domain D_x = [a, b]
      - there exist values in the domains of all the other variables different from x that satisfy c
        - when x = a, and
        - when x = b
  - Weaker than arc-consistency

- Box-consistency: improvement of 2B-consistency
  - Let *c* be a k-ary constraints over variables (x_1, …, x_k), *c* is Box-consistent if for each $x_i \in \{x_1, …, x_k\}$ such that Dx_i = [a, b], the following relationships are satisfied:
    - c(Dx_1, …, Dx_{i-1}, [a, a+), Dx_{i+1}, …,Dx_k)
    - c(Dx_1, …, Dx_{i-1}, (b−, b], Dx_{i+1}, …, Dx_k)

- These algorithms are applied incrementally, when new constraints are acquired
  - Constraints are propagated until reaching a fix point
  - Domains can be reduced or constraints may be left pending
- When domains become empty, the set of constraints is unsatisfiable
- In practice, it is always required to instantiate the variables to ensure the satisfiability of the store

Labelling technique

- Graph of constraints between variables
- Domains of the variables

→ We can instantiate to find (or not) a solution

- Generate and test
  - Generate the values for all the variables
  - Check if the values satisfy the constraints
- Backtracking
  - Generate the values variable by variable
  - Check if the value satisfies the constraints
  - As soon as a variable can not be instantiated change the value of the previous variable

Labelling procedure comparison

- $X \in \{1,2\}$, $Y \in \{1,2\}$, $Z \in \{1,2\}$
- $X \neq Y$, $X \neq Z$, $Y \neq Z$

Generate and test

X	Y	Z	Test
1	1	1	Fail
1	1	2	Fail
1	2	1	Fail
1	2	2	Fail
2	1	1	Fail
2	1	2	Fail
2	2	1	Fail
2	2	2	Fail

Backtracking

X	Y	Z	Test
1	1		Fail
1	2	1	Fail
1	2	2	Fail
2	1	1	Fail
2	1	2	Fail
2	2		Fail

Backtracking techniques are preferred, but where to start from?

- The order of the variables influences the instantiation time
- Various heuristics can be used to select the variables order
  - Most/less constrained
  - Bigger/smaller domains
- Instantiation methods can be improved
  - Look-ahead methods: once a variable is instantiated, the value is propagated through the complete graph using consistency algorithms (AC1)
  - Forward checking: once a variable is instantiated, the value is propagated to its direct neighbours

Libraries for designing constraint solvers

- Prolog (various versions)
- ILOG Solveur (C++) (www.ilog.com/products/solver/),
- JSolver (Java) (www.ilog.com/products/jsolver/),
- Choco (Claire) (www.choco-constraints.net),
- Facile (Ocaml) (www.recherche.enac.fr/opti/facile/),
- CHIP Library (C++) (www.cosytec.fr),
- JCL – Java Constraint Library http://liawww.epfl.ch/JCL/)
- JCK – Java Constraint Kit (http://www.pms.ifi.lmu.de/software/jack/)

Books on Constraint (Logic) Programming

- Programming with Constraints: An Introduction, K. Marriott and P. Stuckey, MIT Press.
- Essentials of Constraint Programming, S. Abdennadher and T. Frühwirth, Springer.

Constraint Logic Programming
Summary

- Constraints solvers are able
  - To manage constraints applying on variables
  - To instantiate constraints satisfaction problems
- Work with finite data domains
  - Eventually, it is necessary to check the existence of solutions to decide the consistency of a set of constraints
- Constraints can be represented by graphs
  - Consistency algorithms propagate constraints
  - Different algorithms → different complexities
  - Trade-off between efficiency of the algorithm and memory space

Constraint Logic Programming
Summary

- Our underlying technology: the CLPS-BZ solver
  - Constraint Logic Programming on Sets for B and Z
- Handles set-theoretical structures (used in B or Z)
  - Sets
  - Relations, functions, injections, bijections, surjections, etc.
- And operators:
  - $=$, $\neq$, $\notin$, $\subseteq$, card, dom, ran, rdom, inv, powerset, $\times$, couple, $\cup$, $\cap$, \
  - other operators have to be rewritten, e.g. $X \subset Y \rightarrow X \subseteq Y \wedge X \neq Y$
- Coupled with CLP(FD) for handling integers
- Arc consistency AC3 algorithm

Outline

1. Notions of Constraint Logic Programming
2. Symbolic animation of models
3. Automated boundary test generation
4. Industrial experience
5. Scenario-Based Testing
6. Conclusions and future work

Symbolic animation of models

Symbolic animation of models

1. A brief introduction to B abstract machines notation
2. Computation of behaviours
3. Principles of symbolic animation
4. Evaluation of behaviours

- B method and notation, introduced by J.-R. Abrial in 1996

- Incremental process of software development based on the notion of refinements

  Abstract machine
  → Refinement 1
  → …
  → Refinement N
  → Implementation → code generation

- Here, only abstract machines are considered

- Formalism based on:
  - A set-theoretical data model
  - First order logics
  - Generalized substitutions

- Contents of a B abstract machine

```
MACHINE(P)
        machine_name
CONSTRAINTS
        Pred(P)
SETS
        S ; T = {a,b}
CONSTANTS
        C      /* constants list */
PROPERTIES
        Pred(P,C)
VARIABLES
        V      /* variables list */
INVARIANT
        Pred(P,C,V)
```

```
INITIALISATION
        Subst(V)

OPERATIONS
   rr ← operation(params) =
        PRE

Pred(P,C,V,params)
        THEN
                Subst(V)
        END ;
```

Symbolic animation of models
The B abstract machines notation

- Generalized substitutions
  - Simple substitutions $x := E$
  - Multiple simple substitutions $x,y := E,F$ $x := E \parallel y := F$
  - Effect-free substitution skip
  - Guarded substitutions IF P THEN S ELSE T END
  - Bounded choice substitutions CHOICE S_1 OR … OR S_N
  - Unbounded choice substitutions *ANY z WHERE P(z) THEN S END*
- Other substitutions can be derived from these

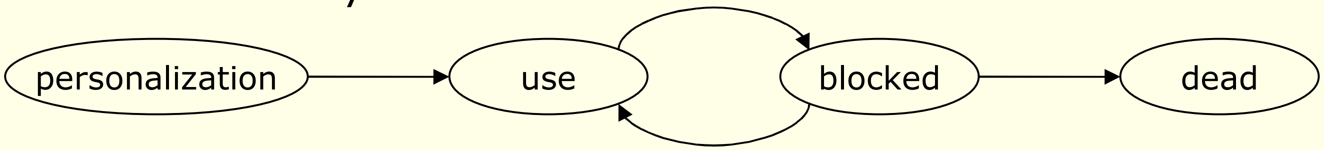
Symbolic animation of models
The B abstract machines notation

- Verification of the coherence a B abstract machine
  - Initialization establishes the invariant
    $\text{Init} \Rightarrow \text{Invariant}$
  - Invariant is preserved by all the operations
    $\text{Invariant} \Rightarrow [\text{OP}]\ \text{Invariant}$

- Other tools (such as ProB) make it possible to do more:
  - Model-checking algorithms by state exploration
  - Detection of deadlocks, etc.
  - Verification of LTL properties

A running example of a B abstract machine

- Model of an electronic purse (as embedded on smart cards)
  - Classical life cycle of a card



  - Two pin codes: holder (3 tries), bank (4 tries)
  - The purse has parameters: user pin code, bank pin code, maximal balance, maximal debit authorized
  - All operations are total and return a status word

- Operations
  - During the personalization:
    - PUT_DATA: personalizes the parameters of the card (maximal balance, maximal debit, holder pin code, bank pin code)
    - STORE_DATA: terminates the personalization phase
  - During the use phase:
    - VERIFY_PIN: checks the PIN on the card
    - INITIALIZE_TRANSACTION: initializes a transaction (debit/credit)
    - COMMIT_TRANSACTION: validates the transaction
  - When the card is blocked
    - PIN_CHANGE_UNBLOCK: changes the holder pin and resets remaining tries

- All the operations can always be invoked
  - but they cannot succeed when invoked in the wrong mode
  - and they return a status word indicating what happened

MACHINE
  Demoney

DEFINITIONS SHORT == 32767..32768; BYTE == -128..127

SETS
  PIN_TYPES = {no_pin, bank, holder} ; CARD_STATUS = {perso, use, blocked, dead} ;
  TRANSACTION_TYPE = {credit, debit}

CONSTANTS
  max_tries_holder, max_tries_bank,
  SET_MAX_BALANCE, SET_MAX_DEBIT, SET_HOLDER_PIN, SET_BANK_PIN

PROPERTIES
  max_tries_holder = 3 ∧max_tries_bank = 4 ∧SET_MAX_BALANCE = 0 ∧
  SET_MAX_DEBIT = 1 ∧SET_HOLDER_PIN = 2 ∧SET_BANK_PIN = 3

VARIABLES
  max_balance, max_debit, holder_pin, bank_pin, balance, transaction, holder_tries,
  bank_tries, mode

INVARIANT
  max_balance ∈ SHORT ∧max_debit ∈ SHORT ∧holder_pin ∈ SHORT ∧bank_pin ∈ SHORT ∧
  balance ∈ SHORT ∧transaction ∈ SHORT ∧mode ∈ CARD_STATUS ∧…

```
INVARIANT
        …
        (mode = perso ⇒ balance = -1 ∧ auth_pin = pin_none) ∧
        (mode ≠ perso ⇒ balance ≥ 0 ∧ max_balance > 0 ∧ max_debit > 0 ∧ holder_pin ∈ 0..9999 ∧
                        max_debit < max_balance ∧ bank_pin ∈ 0..9999 ∧ bank_pin ≠ holder_pin)

INITIALISATION
        mode := perso ‖ balance := -1 ‖ max_balance := -1 ‖
        max_debit := -1 ‖ holder_pin := -1 ‖ bank_pin := -1 ‖
        holder_tries := max_tries_holder ‖ bank_tries := max_tries_bank


OPERATIONS

        sw ← PUT_DATA(p, data) =
                PRE   p ∈ BYTE ∧ data ∈ SHORT
                THEN
                        IF (mode ≠ perso) THEN
                             sw := wrong_mode
                        ELSE
                             IF …
                        END
                END
END
```

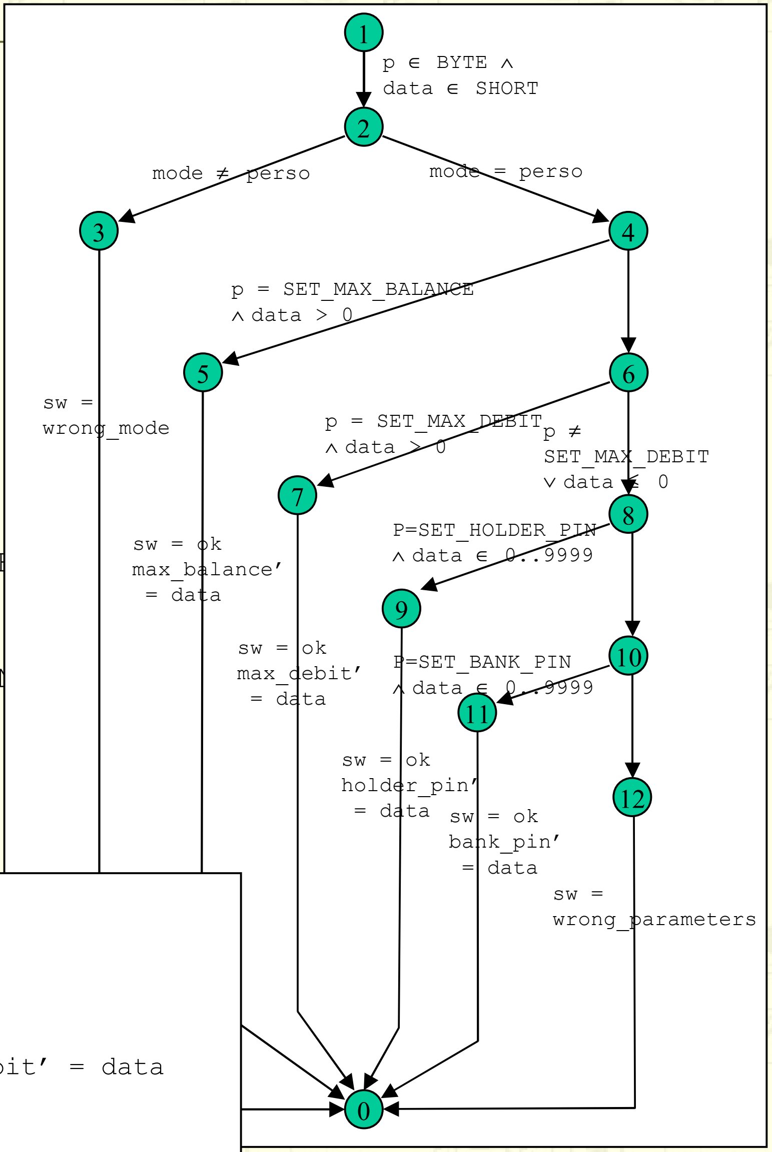
- Behaviours are computed in two steps:
  - Compute the before/after predicate of an operation
  - Compute the Disjunctive Normal Form of Effects

In practice, can be assimilated to computing the paths of the control flow graph of the operations.

Symbolic animation of models

Computation of behaviours


```
sw ← PUT_DATA(p, data) =
  PRE p ∈ BYTE ∧ data ∈ SHORT
  THEN
    IF (mode ≠ perso) THEN
      sw := wrong_mode
    ELSE
      IF (p = SET_MAX_BALANCE ∧ data > 0) THEN
       sw := ok ‖ max_balance := data
     ELSE
       IF (p = SET_MAX_DEBIT ∧ data > 0) THEN
         sw := ok ‖ max_debit := data
       ELSE
         IF (p = SET_HOLDER_PIN ∧ data ∈ 0..9999) THI
          sw := ok ‖ holder_pin := data
         ELSE
          IF (p = SET_BANK_PIN ∧ data ∈ 0..9999) THEN
            sw := ok ‖ bank_pin := data
          ELSE
            sw := wrong_parameters
          END
         END
       END
     END
    END
  END
```



An example of behaviour:

p ∈ BYTE ∧ data ∈ SHORT ∧mode = perso ∧
(p ≠ SET_MAX_BALANCE ∧ data ≤ 0) ∧
p = SET_MAX_DEBIT ∧ data > 0 ∧ sw = ok ∧ max_debit' = data

Denoted [1, 2, 4, 6, 7, 0]

Symbolic animation of models
Symbolic Animation

- Animating a model
  - Semi-automated mean for validating the model
  - Ensures that the model behaves as described in the requirements
  - Different from the verification of the model
- How to animate a model?
  - User selects the operation
  - Instantiates the parameters
  - Tool-support simulates the execution of the transition
- Symbolic animation
  - Improves the possibilities of the animation
  - Makes it possible to abstract parameter values

- Executing a step $s_1 \rightarrow s_2$ is equivalent to solving a CSP between:
  - The values of the state variables in s_1
  - The values of the state variables in s_2
  - The constraints represented by the predicates of the considered behaviour

- Domains of the state variables is given by the machine invariant (we assume that the machine coherence is verified)

- Formally, let:
  - $Inv(s_1)$ be the state predicate characterization of s_1
  - $Inv(s_2)$ be the state predicate characterization of s_2 (in which state variables are primed)
  - $Bhvr(s_1,s_2)$ be the before-after predicate of the considered behaviour

Behaviour Bhvr can be activated if and only if

$Inv(s1) \wedge Inv(s2) \wedge Bhvr(s1,s2)$ is satisfiable

- Input parameters of the operations
  - Can be instantiated by the user
  - Are existentially quantified otherwise

- To animate a B machine, it is mandatory to be able to deal with non-determinism
  - of data: substitutions may introduce unspecified data values
    ANY xx WHERE xx $\in$ 0..10 THEN … END
  - of behaviours: choices guards are not necessarily mutually exclusive

Symbolic animation of models
Symbolic Animation

- Example: *init . PUT_DATA(_X,_Y)*

Initial state

mode = perso
max_balance = -1
max_debit = -1
holder_pin = -1
bank_pin = -1
balance = -1
holder_tries = 3
bank_tries = 4

_R ← PUT_DATA(_X, _Y)

Local variables

p = _X
data = _Y
sw = _R

After state

max_balance = _Y
mode = perso
max_debit = -1
holder_pin = -1
bank_pin = -1
balance = -1
holder_tries = 3
bank_tries = 4

Associated constraints:

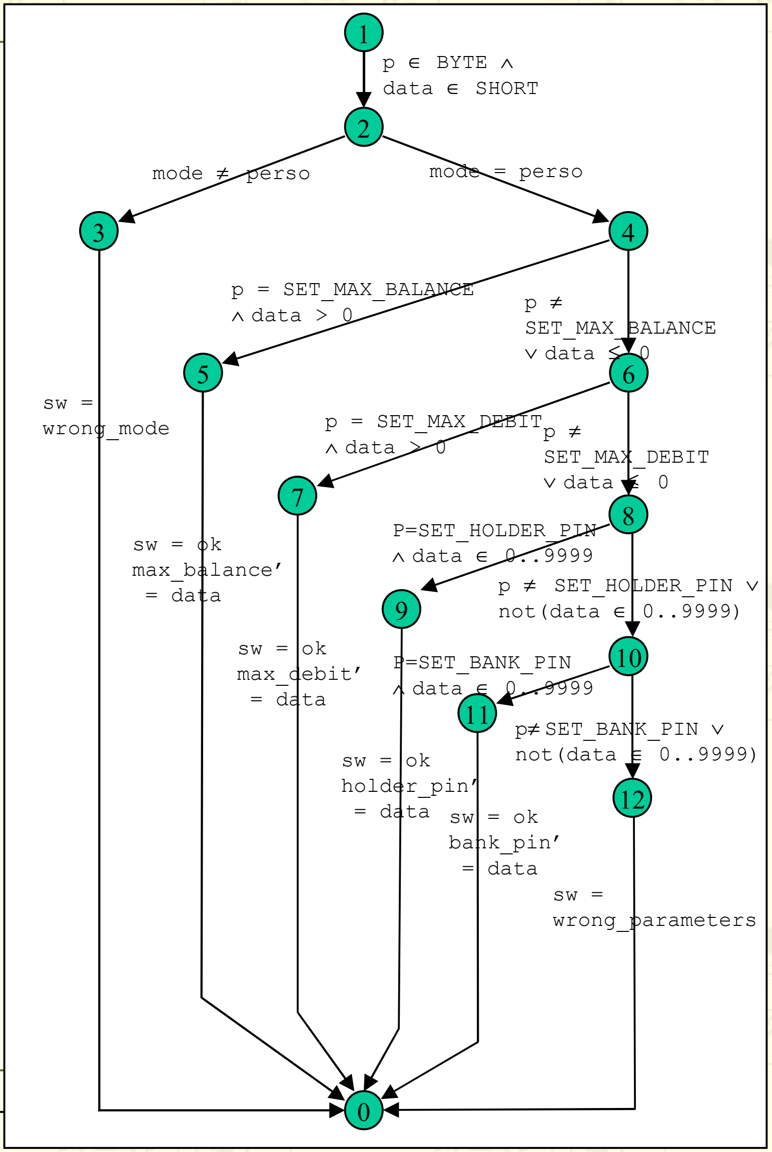
$\_X \in BYTE$

$\_Y \in SHORT$

_X = SET_MAX_BALANCE

$\_Y > 0$

_R = ok



- Example: *init . PUT_DATA(_X,_Y)*

Initial state

mode = perso
max_balance = -1
max_debit = -1
holder_pin = -1
bank_pin = -1
balance = -1
holder_tries = 3
bank_tries = 4

_R ← PUT_DATA(_X, _Y)

Local variables

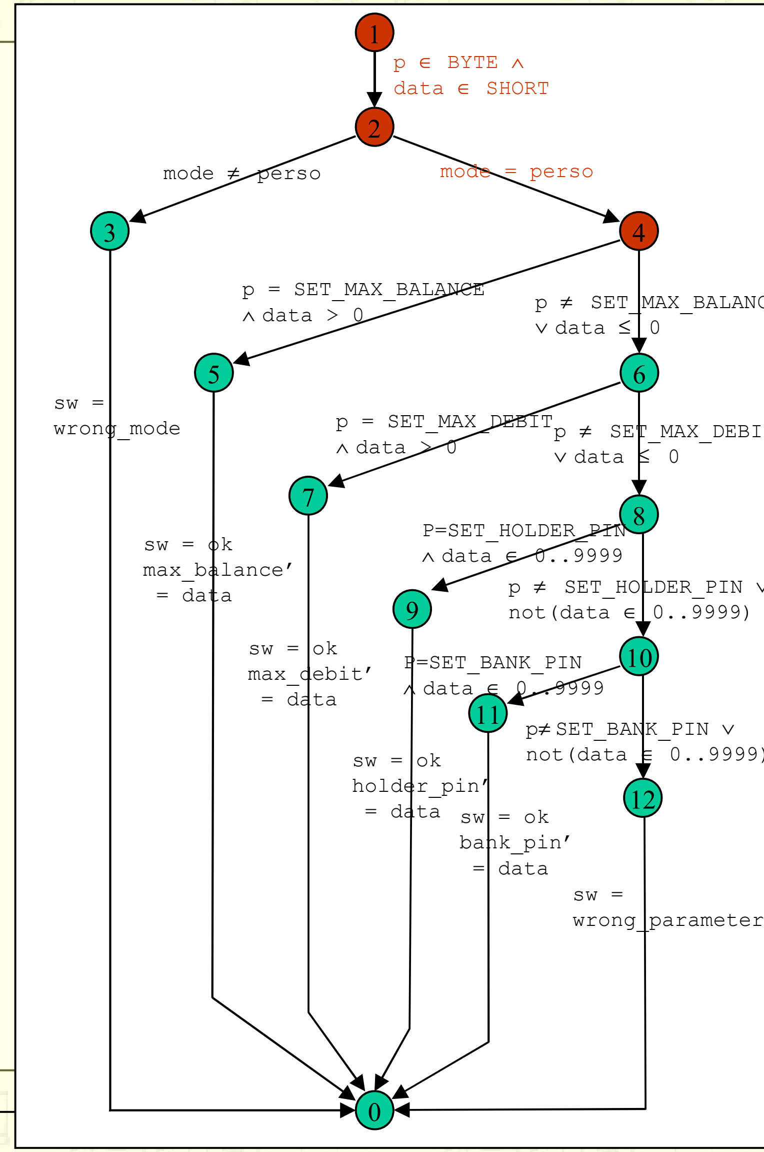
p = _X
data = _Y
sw = _R

After state

max_debit = _Y
mode = perso
max_balance = -1
holder_pin = -1
bank_pin = -1
balance = -1
holder_tries = 3
bank_tries = 4

Associated constraints:

_X ∈ BYTE
_Y ∈ SHORT
_X ≠ SET_MAX_BALANCE ∨ _Y ≤ 0
_X = SET_MAX_DEBIT
_Y > 0
_R = ok



Symbolic animation of models
Symbolic Animation

- Example: *init* . *PUT_DATA(_X,_Y)*

Initial state

mode = perso
max_balance = -1
max_debit = -1
holder_pin = -1
bank_pin = -1
balance = -1
holder_tries = 3
bank_tries = 4

_R ← PUT_DATA(_X, _Y)

Local variables

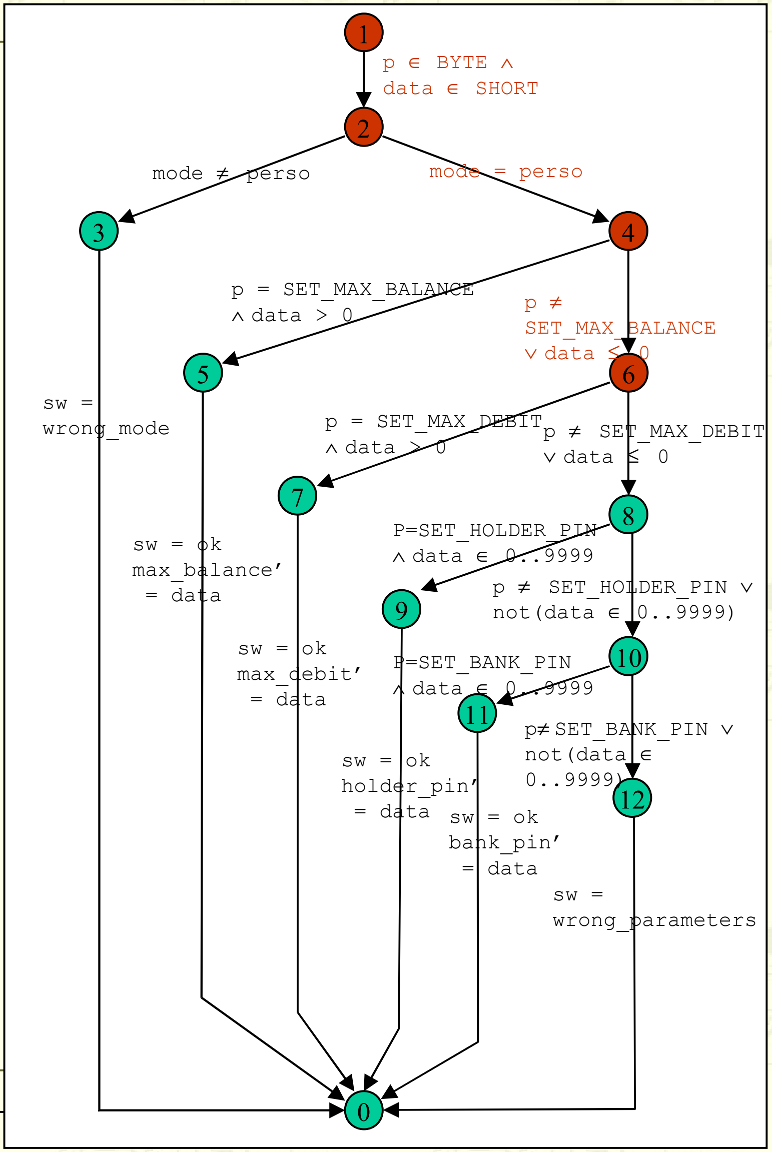
p = _X
data = _Y
sw = _R

After state

holder_pin = _Y
mode = perso
max_balance = -1
max_debit = -1
bank_pin = -1
balance = -1
holder_tries = 3
bank_tries = 4

Associated constraints:

$\_X \in$ BYTE
$\_Y \in$ SHORT
$\_X \neq$ SET_MAX_BALANCE $\vee \_Y \leq 0$
$\_X \neq$ SET_MAX_DEBIT $\vee \_Y \leq 0$
$\_X =$ SET_HOLDER_PIN
$\_Y \in 0..9999$ $\_R =$ ok



- Example: *init . PUT_DATA(_X,_Y)*

Initial state

mode = perso
max_balance = -1
max_debit = -1
holder_pin = -1
bank_pin = -1
balance = -1
holder_tries = 3
bank_tries = 4

_R ← PUT_DATA(_X, _Y)

Local variables

p = _X
data = _Y
sw = _R

After state

bank_pin = _Y
mode = perso
max_balance = -1
max_debit = -1
holder_pin = -1
balance = -1
holder_tries = 3
bank_tries = 4

Associated constraints:

$\_X \in$ BYTE $\_Y \in$ SHORT

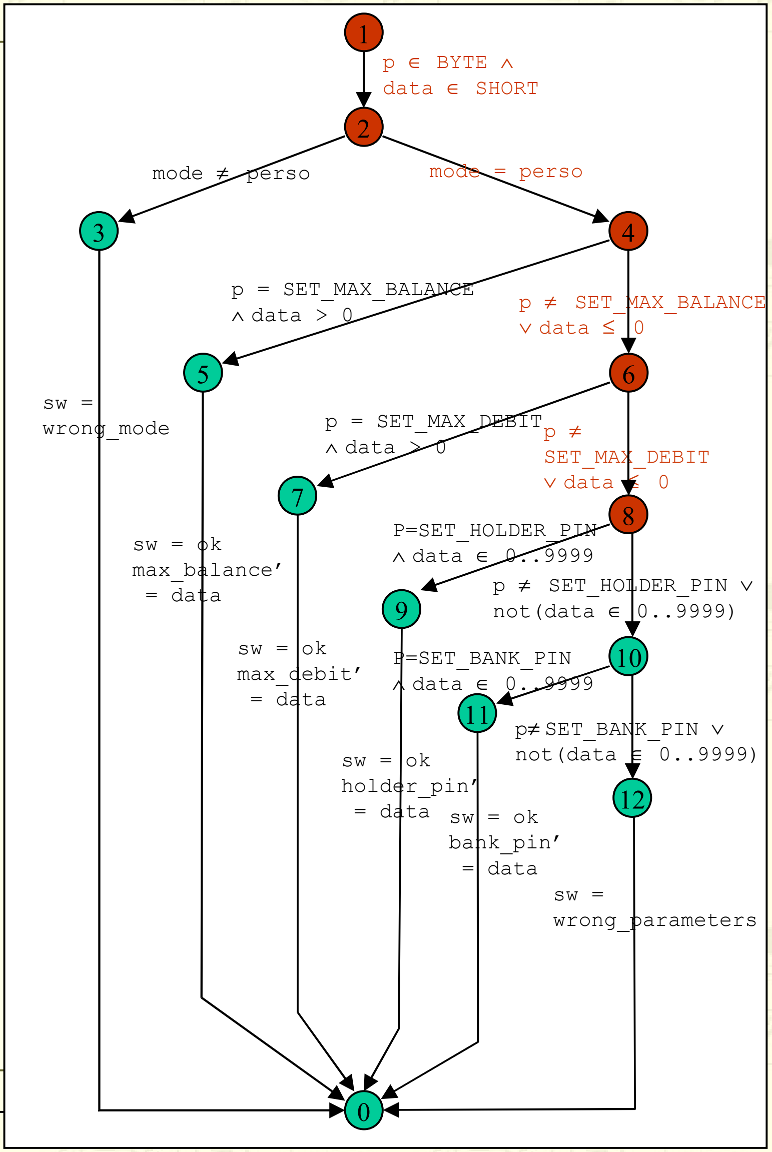
$\_X \neq$ SET_MAX_BALANCE $\vee \_Y \leq 0$

$\_X \neq$ SET_MAX_DEBIT $\vee \_Y \leq 0$

$\_X \neq$ SET_HOLDER_PIN $\vee$ not($\_Y \in$ 0..9999)

_X = SET_BANK_PIN

$\_Y \in$ 0..9999   _R = ok



- Example: *init* . *PUT_DATA(_X,_Y)*

Initial state

mode = perso
max_balance = -1
max_debit = -1
holder_pin = -1
bank_pin = -1
balance = -1
holder_tries = 3
bank_tries = 4

_R ← PUT_DATA(_X, _Y)

Local variables

p = _X
data = _Y
sw = _R

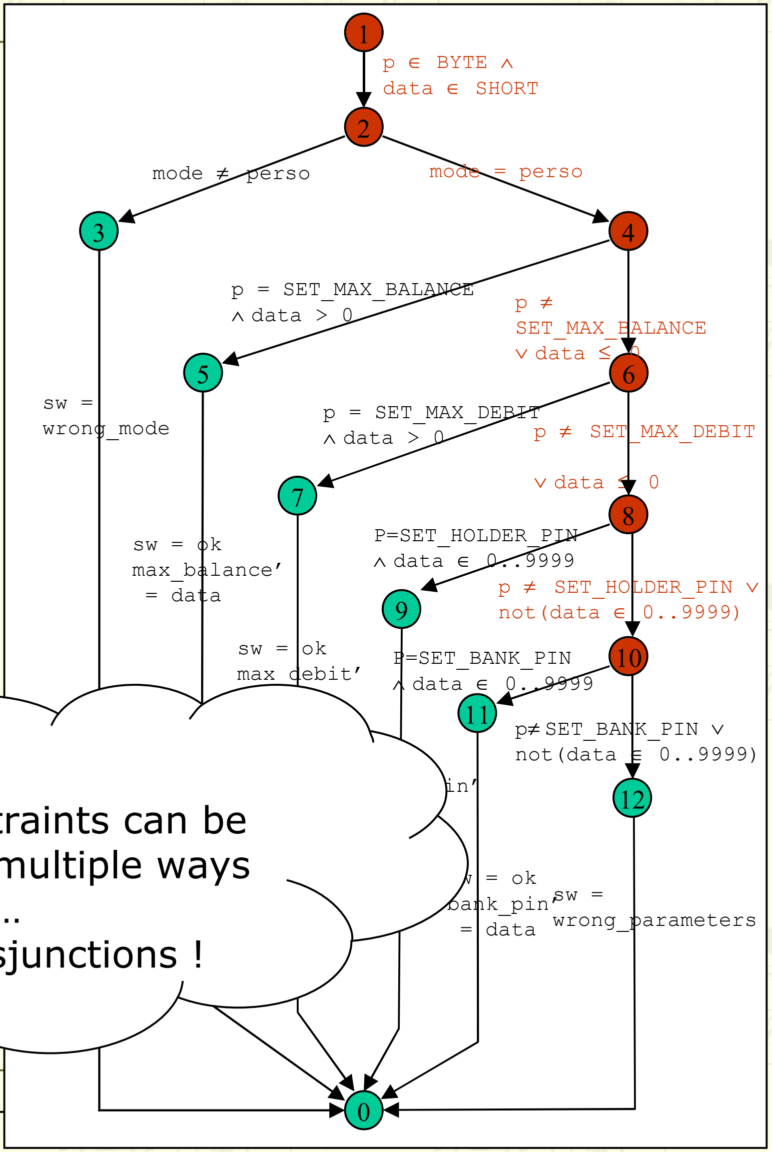
After state

mode = perso
max_balance = -1
max_debit = -1
holder_pin = -1
bank_pin = -1
balance = -1
holder_tries = 3
bank_tries = 4

Associated constraints:

_X ∈ BYTE   _Y ∈ SHORT
_X ≠ SET_MAX_BALANCE ∨ _Y ≤ 0
_X ≠ SET_MAX_DEBIT ∨ _Y ≤ 0
_X ≠ SET_HOLDER_PIN ∨ not(_Y ∈ 0..9999)
_X ≠ SET_HOLDER_PIN ∨ not(_Y ∈ 0..9999)
_R = wrong_parameters

These constraints can be satisfied in multiple ways
…
due to disjunctions !



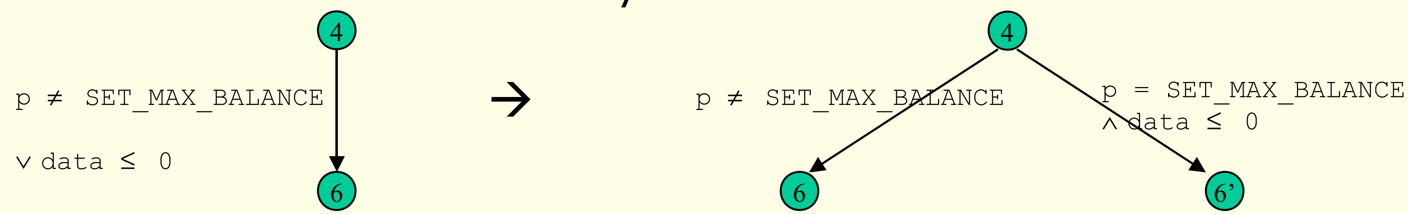
Symbolic animation of models

Computation of behaviours (cont’d)

- Behaviours are computed in two steps:
  - Compute the before/after predicate of an operation
  - Compute the Disjunctive Normal Form of the Effects

In practice, can be assimilated to computing the paths of the control flow graph of the operations.

- Disjunctions create choices when evaluating the predicates
  - Rewritten to simulate a lazy evaluation

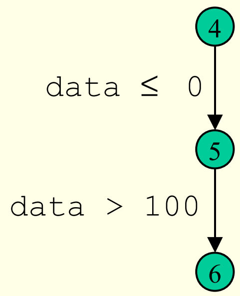


  - Problem of combinatorial explosion and inconsistencies

- Improving the evaluation of specifications
  - Set of modifications applied on the behaviour graphs that help reducing the evaluation time
  - Mainly inspired by compilation techniques

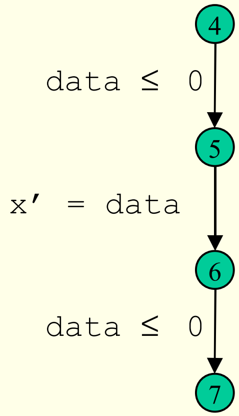
- Modifications performed on the graph
  - Removal of inconsistent paths
  - Removal of common sub-expressions
  - Ordering of atomic predicates
  - Delaying of choice-points

- Removal of inconsistent paths
  - Conjunction on a path is inconsistent



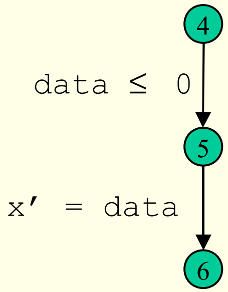
Done by symbolic evaluation

- Removal of common sub-expressions
  - Same predicates appearing twice in a path

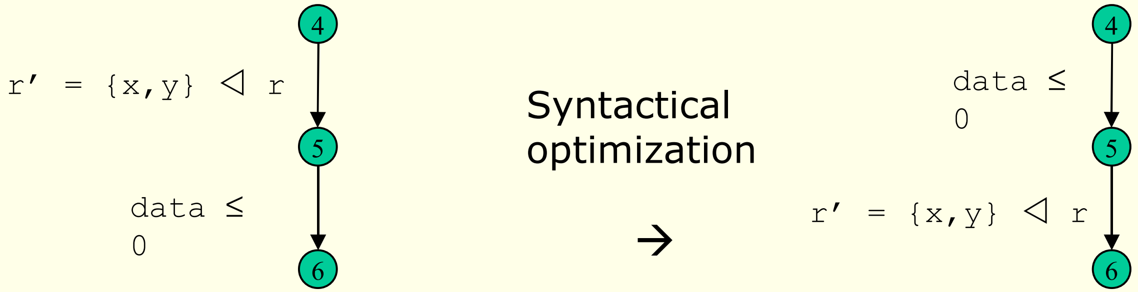


Syntactical optimization

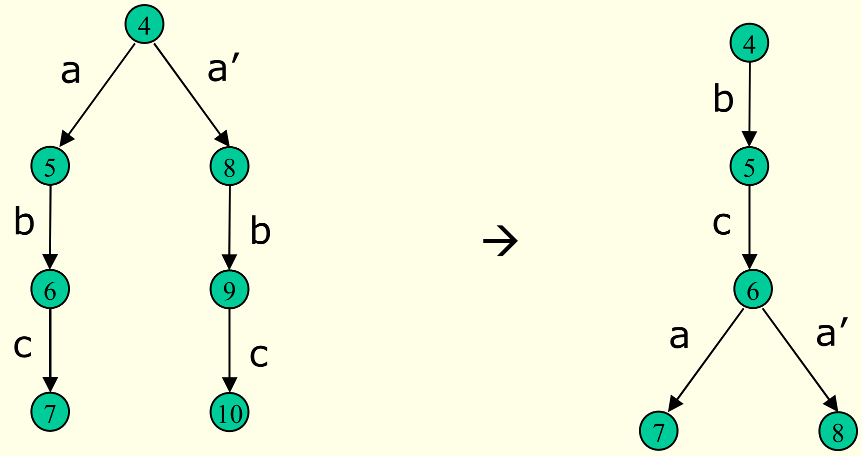
→



- Ordering of atomic predicates
  - Delaying of costly predicates



- Delaying of choice-points
  - Avoids re-evaluation of predicates when backtracking



- Symbolic animation of models is based on the decomposition of operations into behaviours
- Behaviours are activated one-by-one for each step of animation
  - Backtracking is used to iterate over the possible behaviours to be activated
  - An associated constraint store determines the feasibility of the transition (i.e. feasibility of the transition ⇔ consistency of the store)
- Tool-supported process
  - Helps the test architect to validate its test model by testing its dynamics
  - Semi-automated process: manual choice of the operations
- Employed in the test generation process

Outline

1. Notions of Constraint Logic Programming
2. Symbolic animation of models
3. Automated boundary test generation
4. Industrial experience
5. Scenario-Based Testing
6. Conclusions and future work

Automated Boundary Test Generation

A. Automated Boundary Test Generation

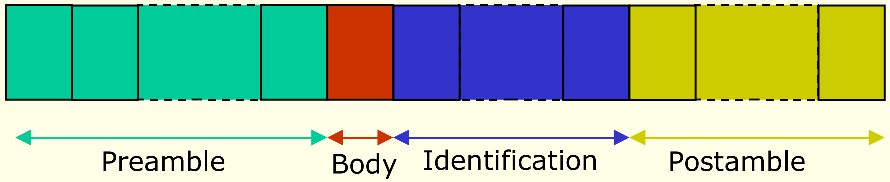
1. General idea of boundary test case generation
2. Computation of the test targets
3. How to reach the test targets?
4. Establishing a conformance verdict

Automated Boundary Test Generation
General idea

- Test a system under test, by using a functional model
  - Test all the behaviours of all the operations
  - Structural coverage of the operations of the system
- Boundary analysis of the data
  - For a given behaviour
  - Model variables (boundary test targets)
  - Operation parameters (boundary parameter values)
- Automated
  - application of structural coverage criteria
  - computation of boundary test targets using CLP
  - computation of a test case

- The SUT can be placed in a state that is equivalent to the initial state of the B machine
- A control point of the SUT can be associated to each operation of the B machine
- Data can be compared: there exists an abstraction function that makes it possible to relate the abstract data of the model and the actual data of the SUT
- The B machine satisfies the classical proof obligations
  - Initialization establishes the invariant
  - All the operations preserve the invariant

A test case is composed of 4 main parts



Preamble: sequence of operations, from the initial state, that reach the test target

Body: activation of the considered behaviour

Identification: call to observation operations

Postamble: sequence of operations that reaches the initial state, or another target

Automated Boundary Test Generation
Computation of the test targets

- Test targets are produced by a boundary analysis
- Concretely, these are states of the model:
  - that make it possible to activate the considered behaviour
  - in which at least one of the state variables is at an extremum (minimum or maximum)

For each operation *op*

postcondition of behaviour 1

$$EDNF(op) = Inv \wedge Pre_1 \wedge Post_1 \; [] \; \ldots \; [] \; Inv \wedge Pre_N \wedge Post_N$$

machine invariant

precondition of behaviour 1

B operator for bounded choice substitutions

- Rewriting rules improving the structural coverage of the model

  - Rewriting rule related to the B semantics:
    IF *P* THEN *S* END → IF *P* THEN *S* ELSE *skip* END

  - Rewriting rules to satisfy decision coverage criteria
    - Applied on disjunctions located in the decision predicates (IF … THEN … ELSE … END)

Rule	**$p \vee q$ becomes:**	**Coverage criterion**
1	$p \vee q$	Condition coverage
2	p [] q	Condition/Decision Coverage
3	$p \wedge \neg q$ [] $\neg p \wedge q$	Modified Condition/Decision Coverage
4	$p \wedge \neg q$ [] $\neg p \wedge q$ [] $p \wedge q$	Multiple Condition Coverage

- One of these rules is selected by the validation engineer (for each operation)
- Inconsistent rewritings are detected by symbolic evaluation and removed

Automated Boundary Test Generation
Computation of the test targets


```
sw ← STORE_DATA =
      BEGIN
         IF mode ≠ perso THEN
            sw := wrong_mode
         ELSE
            IF max_balance = -1 ∨max_debit = -1 ∨holder_pin = -1 ∨bank_pin = -1 THEN
                sw := incomplete_personalization
            ELSE
              IF max_balance > max_debit ∧holder_pin ≠ bank_pin THEN
                 sw := ok ‖ mode := use ‖ balance = 0
              ELSE
                 sw := wrong_personalization
              END
            END
         END
      END
```

Validation engineer wants to test this operation:

- using the MC/DC coverage

→ rewriting of the disjunctions

Automated Boundary Test Generation
Computation of the test targets


```
sw ← STORE_DATA =
    BEGIN
      IF mode ≠ perso THEN
        sw := wrong_mode
      ELSE
        IF max_balance = -1 ∨max_debit = -1 ∨holder_pin = -1 ∨bank_pin = -1 THEN
            sw := incomplete_personalization
        ELSE
          IF max_balance > max_debit ∧holder_pin ≠ bank_pin THEN
            sw := ok ‖ mode := use ‖ balance = 0
          ELSE
            sw := wrong_personalization
          END
        END
      END
    END
```

Target 1:

Inv ∧mode ≠ perso

Automated Boundary Test Generation
Computation of the test targets


```
sw ← STORE_DATA =
    BEGIN
      IF mode ≠ perso THEN
        sw := wrong_mode
      ELSE
        IF max_balance = -1 ∨max_debit = -1 ∨holder_pin = -1 ∨bank_pin = -1 THEN
           sw := incomplete_personalization
        ELSE
          IF max_balance > max_d
             sw := ok ‖ mode := us
          ELSE
             sw := wrong_personal
          END
        END
      END
    END
```

Target 2:
Inv $\wedge$ mode = perso $\wedge$ max_balance = -1 $\wedge$
max_debit $\neq$ -1 $\wedge$ holder_pin $\neq$ -1 $\wedge$ bank_pin $\neq$ -1

Target 3:
Inv $\wedge$ mode = perso $\wedge$ max_balance $\neq$ -1 $\wedge$
max_debit = -1 $\wedge$ holder_pin $\neq$ -1 $\wedge$ bank_pin $\neq$ -1

Target 4:
Inv $\wedge$ mode = perso $\wedge$ max_balance $\neq$ -1 $\wedge$
max_debit $\neq$ -1 $\wedge$ holder_pin = -1 $\wedge$ bank_pin $\neq$ -1

Target 5:
Inv $\wedge$ mode = perso $\wedge$ max_balance $\neq$ -1 $\wedge$
max_debit $\neq$ -1 $\wedge$ holder_pin $\neq$ -1 $\wedge$ bank_pin = -1

```
sw ← STORE_DATA =
      BEGIN
         IF mode ≠ perso THEN
            sw := wrong_mode
         ELSE
            IF max_balance = -1 ∨max_debit = -1 ∨holder_pin = -1 ∨bank_pin = -1 THEN
                sw := incomplete_personalization
            ELSE
              IF max_balance > max_debit ∧holder_pin ≠ bank_pin THEN
                 sw := ok ‖ mode := use ‖ balance = 0
              ELSE
                 sw := wrong_personalization
              END
            END
         END
      END
```

Target 6:
Inv ∧mode = perso ∧max_balance ≠ -1 ∧
max_debit ≠ -1 ∧holder_pin ≠ -1 ∧bank_pin ≠ -1 ∧
max_balance > max_debit ∧holder_pin ≠ bank_pin

Automated Boundary Test Generation
Computation of the test targets


```
sw ← STORE_DATA =
      BEGIN
         IF mode ≠ perso THEN
            sw := wrong_mode
         ELSE
            IF max_balance = -1 ∨max_debit = -1 ∨holder_pin = -1 ∨bank_pin = -1 THEN
                sw := incomplete_personalization
            ELSE
              IF max_balance > max_debit ∧holder_pin ≠ bank_pin THEN
                 sw := ok ‖ mode := use ‖ balance = 0
              ELSE
                 sw := wrong_personaliz
              END
           END
         END
      END
```

Target 7:
Inv ∧mode = perso ∧max_balance ≠ -1 ∧ max_debit ≠ -1 ∧holder_pin ≠ -1 ∧bank_pin ≠ -1 ∧ max_balance > max_debit ∧holder_pin = bank_pin

Target 8:
Inv ∧mode = perso ∧max_balance ≠ -1 ∧ max_debit ≠ -1 ∧holder_pin ≠ -1 ∧bank_pin ≠ -1 ∧ max_balance ≤ max_debit ∧holder_pin ≠ bank_pin

Data coverage

- Possibility offered by constraint solving techniques: perform a boundary analysis of the variables involved in the test target
- For each predicate $Inv \wedge [Pre_i]_{op}$

  (V_1, … , V_k are the state variable involved in Pre_i)
  - BG_i^{max} = maximize($f(V_1, \ldots, V_k)$, ($\exists$ input | $Inv \wedge Pre_i$))
  - BG_i^{min} = minimize($f(V_1, \ldots, V_k)$, ($\exists$ input | $Inv \wedge Pre_i$))
- Optimization functions
  - Domain of any type: $f(x) = 1$
  - Integers: $f = \sum V_i$ or $f = \sum (\sqrt{V_i})$ or $f = \sum (V_i^2)$
  - Sets: $f = \sum \#V_i$ or $f = \sum (\sqrt{\#V_i})$ or $f = \sum (\#V_i^2)$

Considering Target 6:

/* invariant */

max_balance ∈ SHORT ∧max_debit ∈ SHORT ∧holder_pin ∈ SHORT ∧bank_pin ∈ SHORT ∧
balance ∈ SHORT ∧transaction ∈ SHORT ∧mode ∈ CARD_STATUS ∧
(mode = perso ⇒balance = -1 ∧auth_pin = pin_none) ∧
(mode ≠ perso ⇒balance >= 0 ∧max_balance > 0 ∧max_debit > 0 ∧max_debit < max_balance ∧
holder_pin ∈ 0..9999 ∧bank_pin ∈ 0..9999 ∧bank_pin ≠ holder_pin) ∧

/* target context */
mode = perso ∧max_balance ≠ -1 ∧max_debit ≠ -1 ∧holder_pin ≠ -1 ∧bank_pin ≠ -1 ∧
max_balance > max_debit ∧holder_pin ≠ bank_pin

/* no parameters for the operation */

Problem: Unreachable targets

maximize(max_balance+max_debit+holder_pin+bank_pin) ➔ multiple instantiations possible

1. max_balance = 32767, max_debit = 32766, holder_pin = 32767, bank_pin = 32766
2. max_balance = 32767, max_debit = 32766, holder_pin = 32766, bank_pin = 32767

minimize(max_balance+max_debit+holder_pin+bank_pin) ➔ multiple instantiations possible

1. max_balance = -32767, max_debit = -32768, holder_pin = -32768, bank_pin = -32767
2. max_balance = -32767, max_debit = -32768, holder_pin = -32767, bank_pin = -32768

Automated Boundary Test Generation
Computation of the test targets (cont'd)

Targets will not be reachable:

- Invariant is too weak
- Precondition of the effect is too weak

Which one has to be changed?

→ The invariant shall capture at best the reachable states of the system
→ If everything can not be expressed within the invariant, also strengthen the preconditions

Modification of the invariant: addition of the following predicates:

max_balance ≥ -1 ∧ max_balance ≠ 0 ∧ max_debit ≥ -1 ∧ max_debit ≠ 0 ∧ holder_pin ≥ -1 ∧ holder_pin ≤ 9999 ∧ bank_pin ≥ -1 ∧ bank_pin ≤ 9999

Re-considering Target 6:

/* invariant */
max_balance ∈ SHORT ∧max_debit ∈ SHORT ∧holder_pin ∈ SHORT ∧bank_pin ∈ SHORT ∧
balance ∈ SHORT ∧transaction ∈ SHORT ∧mode ∈ CARD_STATUS ∧
(mode = perso ⇒balance = -1 ∧auth_pin = pin_none) ∧

/* additional part of the invariant */
max_balance ≥ -1 ∧max_balance ≠ 0 ∧max_debit ≥ -1 ∧max_debit ≠ 0
holder_pin ≥ -1 ∧holder_pin ≤ 9999 ∧bank_pin ≥ -1 ∧bank_pin ≤ 9999

/* target context */
mode = perso ∧max_balance ≠ -1 ∧max_debit ≠ -1 ∧holder_pin ≠ -1 ∧bank_pin ≠ -1
max_balance > max_debit ∧holder_pin ≠ bank_pin

Reachable targets

maximize(max_balance+max_debit+holder_pin+bank_pin) ➔ multiple instantiations possible

1. max_balance = 32767, max_debit = 32766, holder_pin = 9999, bank_pin = 9998
2. max_balance = 32767, max_debit = 32766, holder_pin = 9998, bank_pin = 9999

minimize(max_balance+max_debit+holder_pin+bank_pin) ➔ multiple instantiations possible

1. max_balance = 2, max_debit = 1, holder_pin = 0, bank_pin = 1
2. max_balance = 2, max_debit = 1, holder_pin = 1, bank_pin = 0

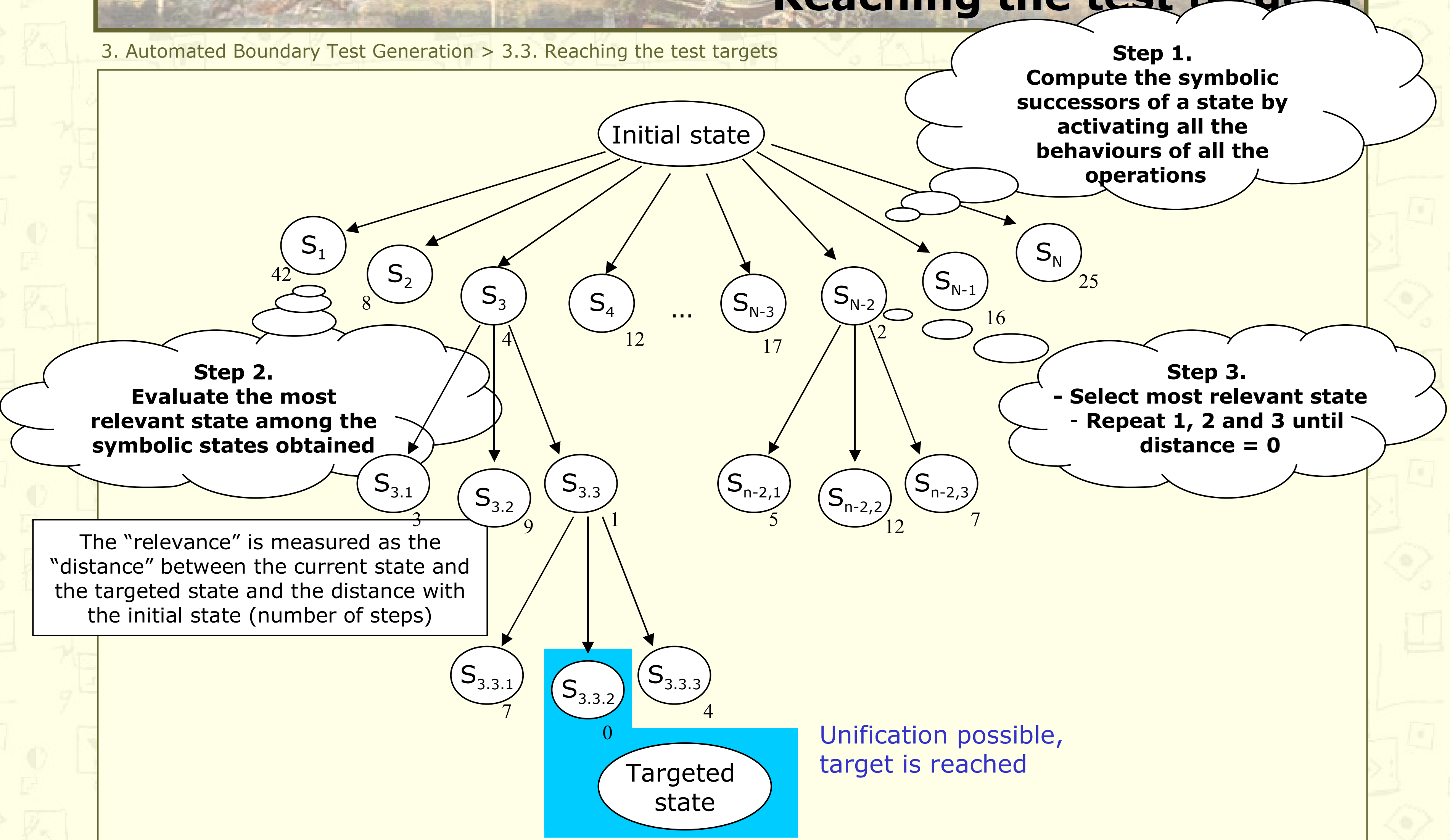
Automated Boundary Test Generation
Reaching the test targets

- Each boundary test target has to be covered by a test

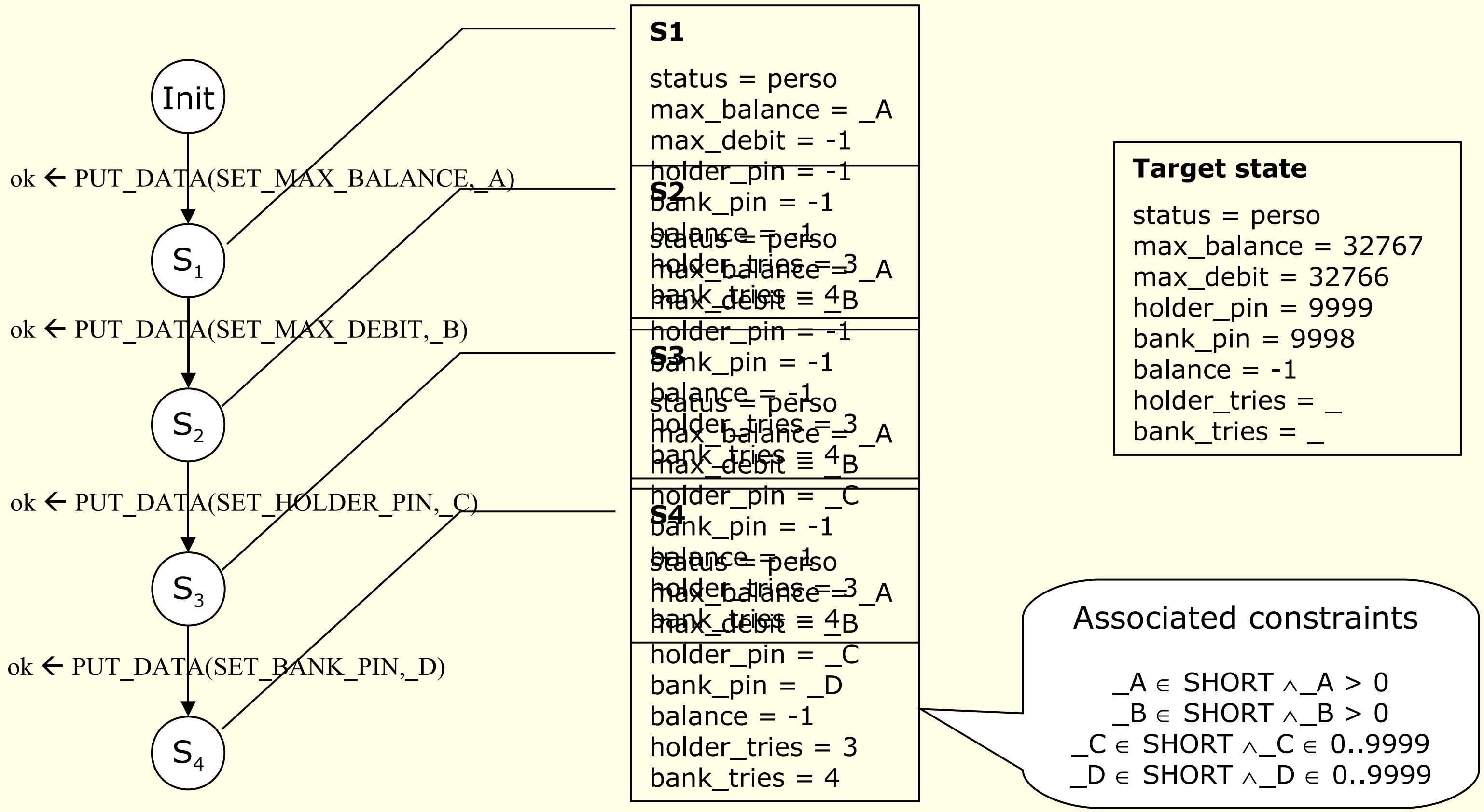
- Idea: automated computation of the preamble
  - Exploration of the state space
  - Use of the symbolic animation to improve the computation
  - Advantage: symbolic states are visited (instead of concrete states)
  - Drawback: impossibility to detect symbolic states already visited

- Best-first algorithm, variant of the A* algorithm
  - Bounded in depth
  - Breadth-first algorithm
  - Improved by a heuristic that “helps” converging to the target

Automated Boundary Test Generation
Reaching the test targets



Re-considering Target 6, maximized:



Automated Boundary Test Generation
Problems of unreachable targets

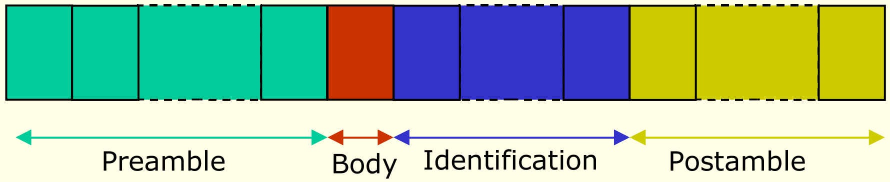
- Best-first algorithm is efficient in practice, despite a $O(x^d)$ complexity
  - x: number of behaviours existing in the system
  - d: search depth

- But may not be able to build a preamble/postamble:
  - Because the target is not reachable
    → Should be prevented by strengthening the invariant
  - Because the depth search is too small
    → What can we do?

- Solution 1: build the preamble by hand ☹
  - Mechanism of "preamble helper": piece of execution sequence built using the animation of the model to reach a specific target

- Solution 2: change the initial state
  - Do not consider the initial state of the machine, but a user-defined one
  - Impacts the concretization of the tests

A test case is composed of 4 main parts



Preamble: sequence of operations, from the initial state, that reach the test target

Body: activation of the considered behaviour

Identification: call to observation operations

Postamble: sequence of operations that reaches the initial state, or another target

Automated Boundary Test Generation
The problem of observation

- Conformance relationship is given by the observable outputs
  - in the preamble
  - in the body
  - in the observations
- The quality of the test verdict is directly related to the number of observation points
- The model has to present observation operations
  - Preferably effect-free operations
- The user is asked to define by hand the calls to observation operations

Automated Boundary Test Generation
Summary

- Each operation is tested for each of its behaviours
- Test targets are extracted from a boundary analysis of the behaviours activation conditions
- Test cases are composed of 4 parts
  - Preamble: computed automatically using symbolic animation
  - Body: activation of the considered behaviour
  - Identification: user-defined calls to observation operations
  - Postamble: optional part, supposed to return to the initial state
- Implemented within the BZ-Testing-Tools framework
  - Experimented in various case studies with various industrial partners
  - Later exported to Leirios Technologies company as LTG-B

Outline

1. Notions of Constraint Logic Programming
2. Symbolic animation of models
3. Automated boundary test generation
4. Industrial experience
5. Scenario-Based Testing
6. Conclusions and future work

Industrial experience

Industrial experience

1. Partnerships and experimental results
2. The Leirios Technologies/Smartesting experience
3. Demo of the Leirios Test Generator
4. A word on Test Designer for UML/OCL

Some industrial partners, over the years …

ROLEX

CB

DASSAULT SYSTEMES

SNCF

france telecom

BNP PARIBAS

PSA PEUGEOT CITROËN

ingenico

Sagem Communication
Groupe SAFRAN

esa

Atos Origin

gemalto
security to be free

PARKEON

AUSTRIACARD
read the future

Some (old) case studies …

- **GSM 11-11 Standard** – SchlumbergerSema/Smart Card Montrouge - 99/00
- **Algorithm for the validation of Metro/RER tickets** – SchlumbergerSema/e-City Besançon - 00/01
- **Java Card Transaction Mechanism** – SchlumbergerSema/Smart Card Montrouge - 01/02
- **“Generic Visibility” module controller** – Peugeot Citroën Automobiles La Garennes-Colombes- 02

And many more … since the creation of the Leirios Technologies company

GSM 11-11 Standard – First evaluation of the test generation method

- Norm describing
  - The interface between a SIM card and the Mobile Environment
  - The logical structure of the SIM
  - The functionalities of the SIM
  - The security in the data access

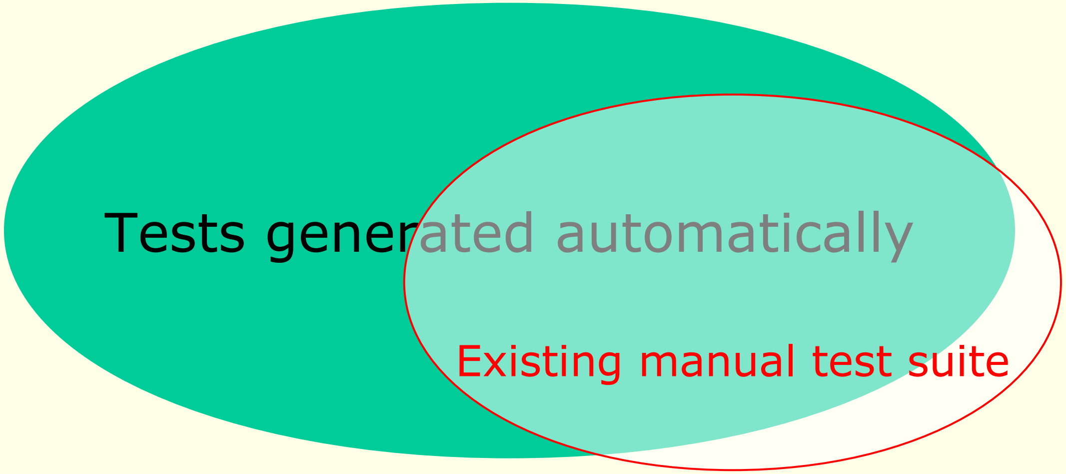
- Case study aiming at:
  - Validating the security aspects of the card
  - Generating abstract test cases
  - The relevance of the method w.r.t. manual testing campaigns

GSM 11-11 Standard – First evaluation of the test generation method

Some metrics:

- 12 pages of B specifications
- 11 operations
- 16 state variables
- 1000 test cases

GSM 11-11 Standard – First evaluation of the test generation method



- Tests completing the existing test suite (50%)
- Tests in common (85% of the existing test suite)
- Tests of the existing test suite not generated (15%)

A comparison between manual/automated process

Manual process		**BZ-Testing-Tools process**	
Test design	20 m/d	B model design	15 m/d
Design of the test plan	5 m/d	Test generation	auto
Test execution	5 m/d	Test execution	5 m/d
Total	30 m/d	Total	20 m/d

Industrial experience
Leirios Technologies/Smartesting

- BZ-Testing-Tools project funded by the ANVAR
  - Agence Nationale pour la Valorisation de la Recherche/National agency for the promotion of research activities
- 2003 – Creation of the Leirios Technologies company
  - Technology transfer in partnership with the university
  - Main product: Leirios Test Generator for B machines
  - 20-30 employees, including former PhD students
  - Improvement of the existing interface (requirements traceability)
  - Consulting activities
- 2008 – Leirios Technologies becomes Smartesting
  - Change of modeling language: B abandoned, replaced by UML/OCL
  - Main product: Test Designer for UML/OCL
  - Increased consulting activities
- http://www.smartesting.com

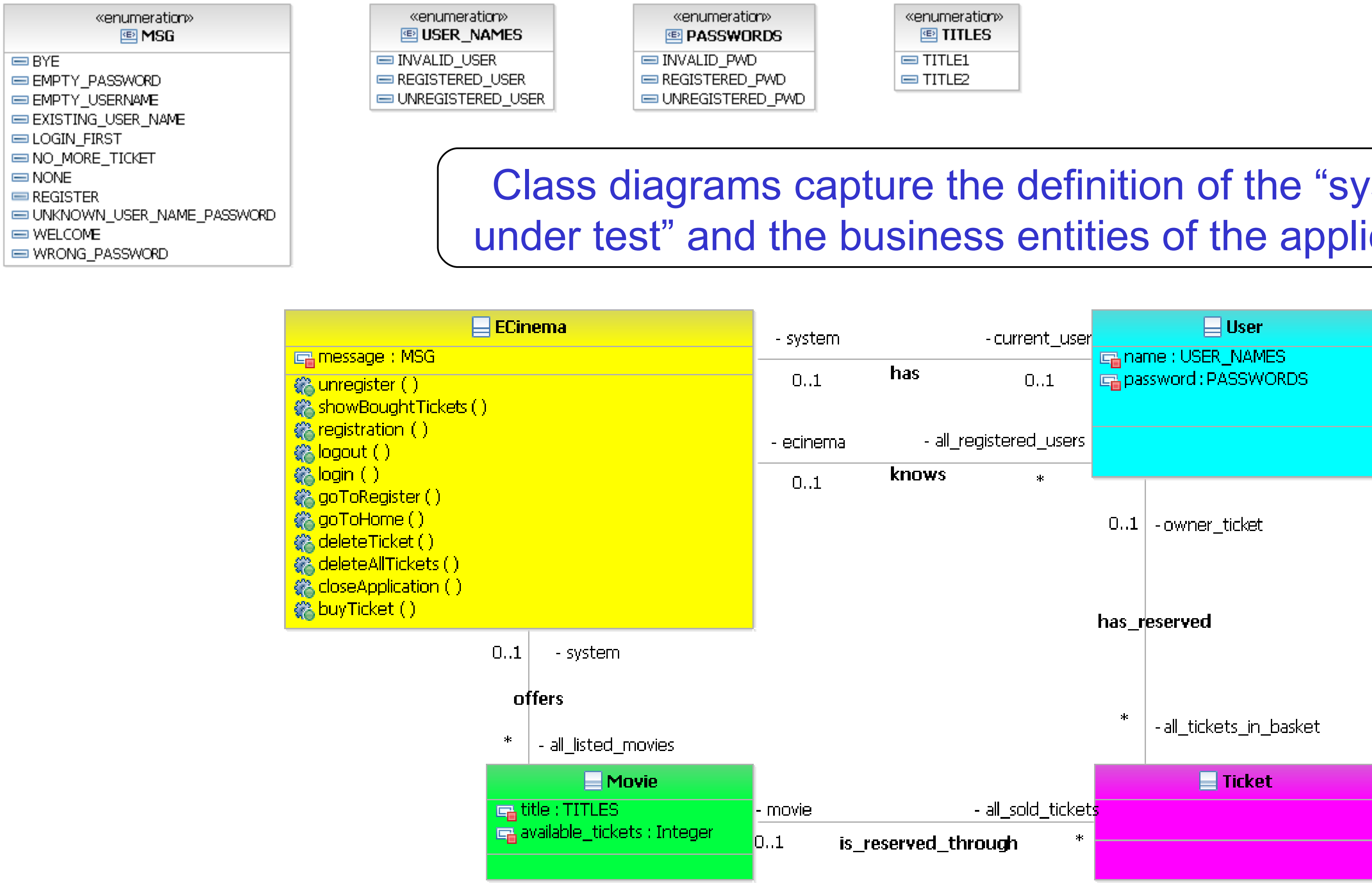
Demo of the Leirios Test Generator for B

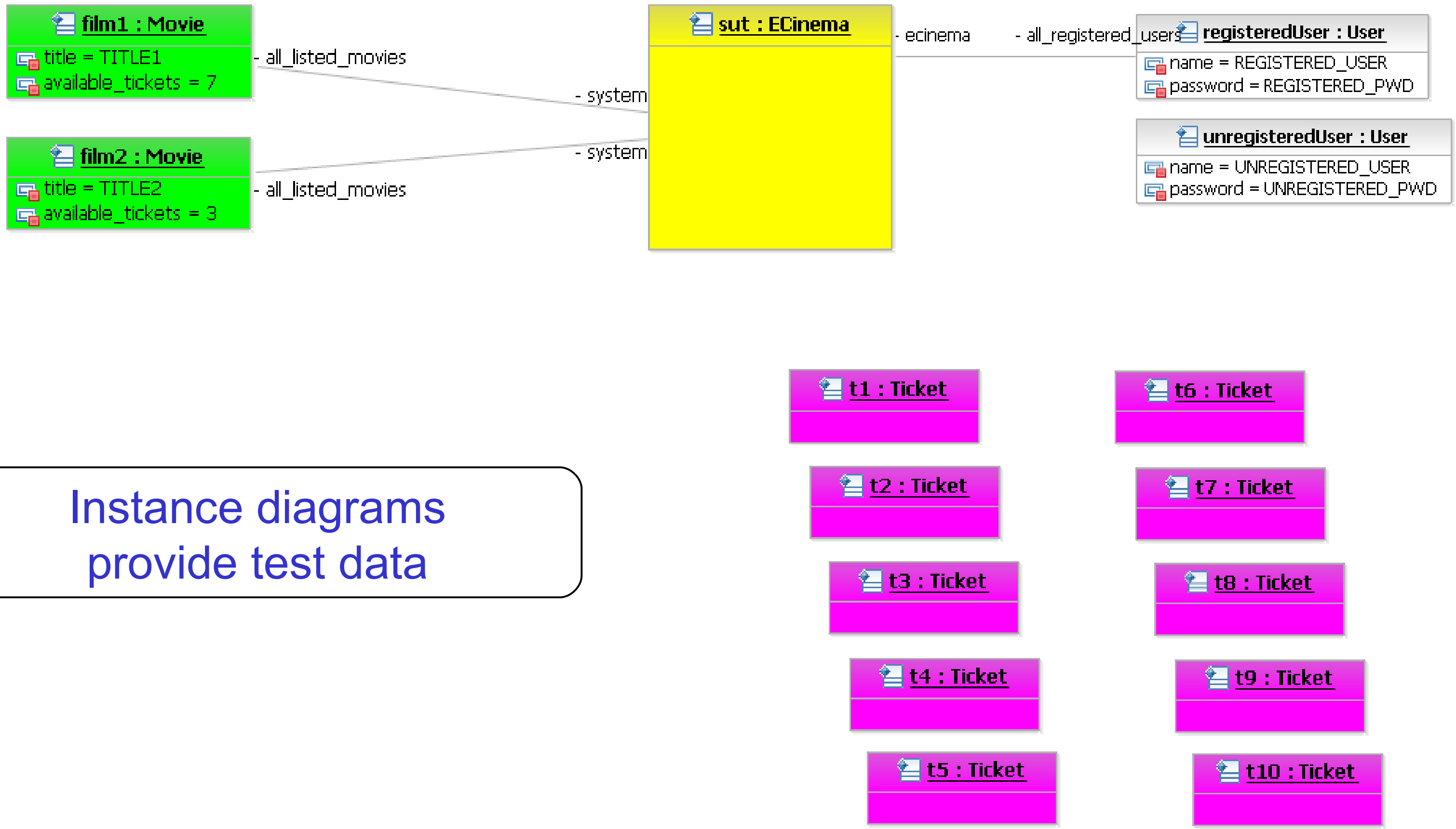
Industrial experience
A word on Test Designer for UML

- Forget about the complexity of the previous version LTG-B
  - No more complicated language
  - No more complicated interface
  - Just imagine a big green button to run the test generation …

- What happened? Lessons from the LTG experience …
  - B is expressive and has a well-defined semantics, but difficult to learn
  - UML is industrial-friendly
  - Nevertheless, OCL constraints are taken into account (OCL is seen as an action language, with conditions but no loops)
  - Target clients have changed → focus on information systems
  - Interface was too complicated: required an expert on the tool to use it → simplified interface

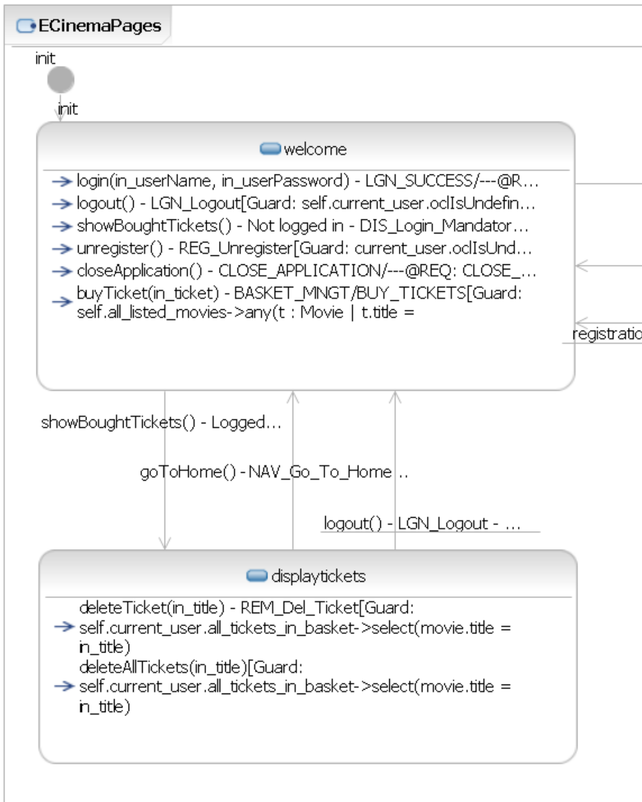
Class diagrams capture the definition of the "system under test" and the business entities of the application





Instance diagrams provide test data

OCL constraints capture business rules, and are used to evaluate transitions



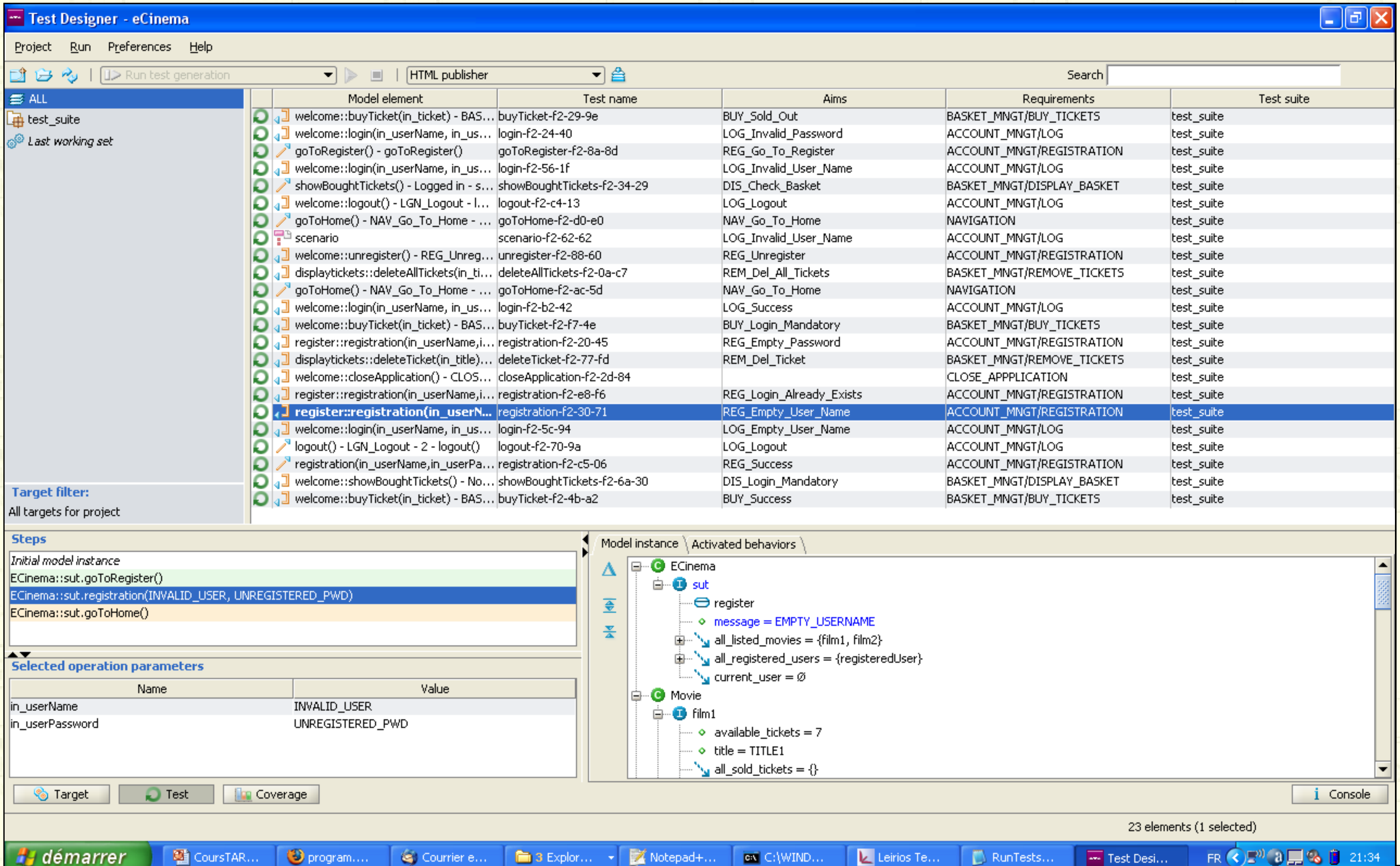

```
context login(in_userName,in_userPassword)::effect:

---@REQ: ACCOUNT_MNGT/LOG
if in_userName = USER_NAMES::INVALID_USER then
   ---@AIM: LOG_Empty_User_Name
   message= MSG::EMPTY_USERNAME
else
   if not all_registered_users->exists(name = in_userName) then
      ---@AIM: LOG_Invalid_User_Name
      message= MSG::UNKNOWN_USER_NAME_PASSWORD
   else
      let user_found:User = all_registered_users->any(name = in_userName) in
         if user_found.password = in_userPassword then
            ---@AIM: LOG_Success
            self.current_user = user_found and
            message = MSG::WELCOME
         else
            ---@AIM: LOG_Invalid_Password
            message = MSG::WRONG_PASSWORD
         endif
   endif
endif
```

behavior of the system

Industrial experience
A word on Test Designer for UML

- Test targets are automatically computed as transitions
  - in the statechart
  - in the OCL code of the operations
- No more boundary analysis
- Improved traceability of requirements and test targets
- No more observation operations
  - Abstract test cases contain the values of variables and outputs
  - Observations are done at concretization-time

Industrial experience
A word on Test Designer for UML



Outline

1. Notions of Constraint Logic Programming
2. Symbolic animation of models
3. Automated boundary test generation
4. Industrial experience
5. Scenario-Based Testing
6. Conclusions and future work

1. Scenario-Based Testing
    1. Principles and motivations
    2. Scenario description language
    3. Unfolding of the scenarios

Fact:

Fully automated testing is nice, but some limitations remain …

- Unreached test targets
  - Preamble computation may fail
- Limited observation points
  - Few observation points implies a less accurate verdict
  - Need for dynamic observation (build an observation sequence that will illustrate if the test went right or wrong)
- Does not cover the dynamics of the system
  - Can be encoded in the model, but requires skills to drive the test generation with appropriate options

Idea:

Give the validation engineer a possibility to write his own tests cases, but let's use our technology to simplify this task

- What are scenarios?
  - Succession of steps, seen as sequences of operations
  - Each step is justified by a specific purpose
- Something new?
  - Not necessarily …
    - Combinatorial testing tools such as Tobias
    - UML-based tools working on sequence diagrams
    - Test purposes of TGV/STG
  - … but it improves existing works!
    - Coupling with constraint logic programming avoids a complete specification of the test cases

- How to design the scenarios?
  - Expressed using regular expressions describing
    - Sequences of operations (without parameters)
    - Intermediate states that have to be reached
  - Textual description of the test cases

- How does it work?
  - Regular expression is unfolded and played on the model using symbolic animation
  - Systematic consistency checks prune incoherent sequences
  - Backtracking makes it possible to iterate over the different solutions

- **Sequence layer**: describes how operations are chained

  Seq ::= Op
  | Seq "." Seq
  | Seq Repeat
  | Seq Choice Seq
  | Seq "→(" Predicate ")"

  Repeat ::= "?" | n | m..n

- **Model layer**: draws the link between the scenario and the model

  Op ::= Op_driven | Op1

  Predicate ::= state_predicate

  Op1 ::= operation_name | "$OP" | "$OP" "\" ListOp

- **Directive layer**: test driving (optimization of unfolding)

  Op_driven ::= "[" Op1 "]" | "[" Op1 "/w" ListB "]" | "[" Op1 "/e" ListB "]"
  ListOp ::= "{" operation_name ("," operation_name)* "}"
  ListB ::= "{" behavior_label ("," behavior_label)* "}"
  Choice ::= "|" | "⊕"

Scenario-Based Testing
Scenario Description Language

- From very detailed scenario …

  [PUT_DATA /w {ok}]^{4} . STORE_DATA →(mode=use) . VERIFY_PIN^{3} →(mode=blocked)

- To less detailed scenarios

  \$OP$^{0..6}$ →(mode=use) . \$OP$^{0..6}$→(mode=blocked)

- We can now exercise the dynamics of the system
  "A failure in the authentication forgets the previous authentication"

  … . [VERIFY_PIN /w {ok,ko}]$^{1..3}$. [INIT_TRANSACTION /w {credit}]

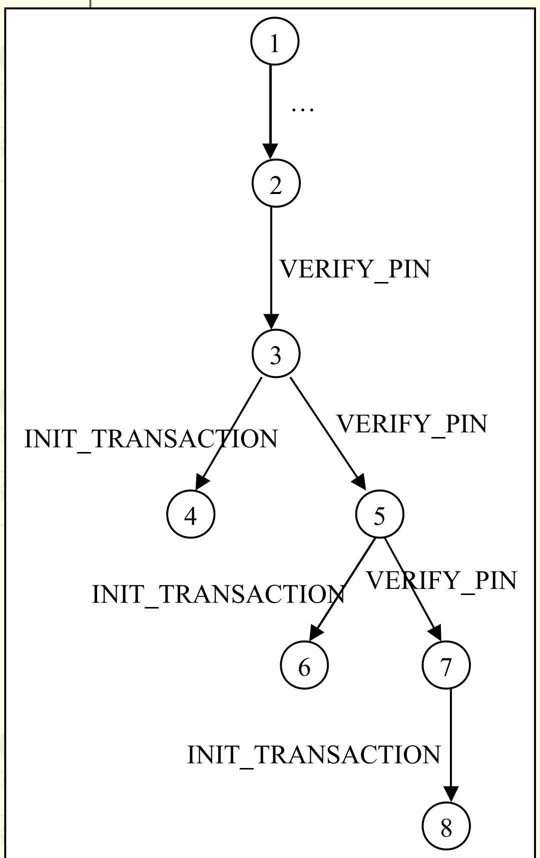
… . [VERIFY_PIN /w {ok,ko}]$^{1..3}$. [INIT_TRANSACTION /w {credit}]

```
sw ← VERIFY_PIN(pin,data) =
  PRE
    pin ∈ PIN_TYPE ∧ data ∈ SHORT
  THEN
    IF (mode = use ∧ pin = holder) THEN
      IF (data = holder_pin) THEN
         sw := ok ‖ auth_pin := holder ‖ holder_tries := max_holder_tries   /* @ok */
      ELSE   /* ko */
        holder_tries := holder_tries - 1 ‖ auth_pin := none ‖
        IF (holder_tries = 1) THEN
          sw := blocked ‖ mode := blocked   /*
        ELSE
          sw := wrong_pin  /* @ko */
        END
      …
    END
```

```
sw ← INITIALIZE_TRANSACTION(type,data) =
   PRE
      type ∈ TRANSACTION_TYPE ∧ data ∈ SHORT
   THEN
      IF (mode = use) THEN
         IF (type = credit ∧ data > 0) THEN
            /* @credit */
            IF (auth_pin = holder_pin) THEN
                sw := ok ‖ transaction = data
            ELSE
                sw := holder_not_authentified
            END
       …
  END
```

Scenario-Based Testing
Unfolding the scenarios

… . [VERIFY_PIN /w {ok,ko}]$^{1..3}$. [INIT_TRANSACTION /w {credit}]



Test sequences :

… VERIFY(holder,$\_P_1$) ; INIT(credit, _X)

… VERIFY(holder,$\_P_2$) ; INIT(credit, _X)

… VERIFY(holder,$\_P_1$) . VERIFY(holder,$\_P_1$) . INIT(credit, _X)

… VERIFY(holder,$\_P_1$) . VERIFY(holder,$\_P_2$) . INIT(credit, _X)

… VERIFY(holder,$\_P_2$) . VERIFY(holder,$\_P_1$) . INIT(credit, _X)

… VERIFY(holder,$\_P_2$) . VERIFY(holder,$\_P_2$) . INIT(credit, _X)

… VERIFY(holder,$\_P_1$) . VERIFY(holder,$\_P_1$) . VERIFY(holder, $\_P_1$) . INIT(credit, _X)

… VERIFY(holder,$\_P_1$) . VERIFY(holder,$\_P_1$) . VERIFY(holder, $\_P_2$) . INIT(credit, _X)

… VERIFY(holder,$\_P_1$) . VERIFY(holder,$\_P_2$) . VERIFY(holder, $\_P_1$) . INIT(credit, _X)

… VERIFY(holder,$\_P_1$) . VERIFY(holder,$\_P_2$) . VERIFY(holder, $\_P_2$) . INIT(credit, _X)

… VERIFY(holder,$\_P_2$) . VERIFY(holder,$\_P_1$) . VERIFY(holder, $\_P_1$) . INIT(credit, _X)

… VERIFY(holder,$\_P_2$) . VERIFY(holder,$\_P_1$) . VERIFY(holder, $\_P_2$) . INIT(credit, _X)

… VERIFY(holder,$\_P_2$) . VERIFY(holder,$\_P_2$) . VERIFY(holder, $\_P_1$) . INIT(credit, _X)

With constraints:

X > 0, $\_P_1$ = holder_pin, $\_P_2 \in 0..9999$, $\_P_2 \neq$ holder_pin

- Prospective research done at LIFC upstream from Smartesting

- Semi-automated test generation process
  - User-defined regular expressions mixing operation calls and states to reach
  - Unfolded and instantiated by symbolic animation

- Advantages of the Scenario Based Testing approach
  - Avoids the validation engineer to compute the appropriate test data
  - Ensures the coverage of certain properties (specified as scenarios)
  - Provides an immediate traceability of the test cases
  - Makes it possible to complement the automated functional test generation
  - Employed successfully in a national project (with Gemalto)

- Drawback
  - Combinatorial testing!

Outline

1. Notions of Constraint Logic Programming
2. Symbolic animation of models
3. Automated boundary test generation
4. Industrial experience
5. Scenario-Based Testing
6. Conclusions and future work

Conclusion and on-going work

1. Conclusion and on-going work

    1. Conclusion
    2. On-going work
    3. Related research papers

We have seen:

- Notions of constraint logic programming …
- … that, coupled with the symbolic animation of behavioural models …
- … make it possible to generate complete test cases …
- … as implemented in the software testing solution of Smartesting
- … but can also be employed for scenario based testing techniques

Conclusion and on-going work
What we do now …

- Testing for Evolution & Security
  - European project FP7 SecureChange http://www.securechange.eu
  - Considering evolutions at each step of the software life cycle
  - Design of security-specific test cases

- Further work on the test scenarios, some leads
  - Improving the scenario language to make it less abstract
    - Adding control structures (conditions, loops) in the scenario description language
    - Goal: get closer to Parameterized Unit Tests of Microsoft
  - Improving the scenario language treatment to deal with abstract scenarios
    - Automated generation of scenarios from properties
  - Adapting scenario-based techniques to UML/OCL
    - In collaboration with Smartesting
  - Definition of “test intention”

- Symbolic animation of models
  - F. Bouquet, B. Legeard, and F. Peureux. **CLPS-B: A Constraint Solver to Animate a B Specification.** International Journal on Software Tools for Technology Transfer, STTT, 6(2):143--157, 2004.
  - F. Bouquet, F. Dadeau, and B. Legeard. **Using Constraint Logic Programming for the Symbolic Animation of Formal Models.** Procs of the Int. Workshop on Constraints in Formal Verification (CFV'05), Tallinn, Estonia, 2005.

- Optimizations on behaviours computation
  - F. Bouquet, B. Legeard, M. Utting, and N. Vacelet. **Faster Analysis of Formal Specification.** 6th Int. Conf. on Formal Engineering Methods (ICFEM'04), volume 3308 of LNCS, Seattle, WA, 2004.

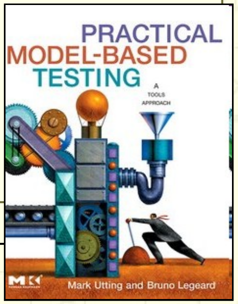
- Automated boundary test generation
  - B. Legeard, F. Peureux, and M. Utting. **Automated boundary testing from Z and B**. In Proc. of the Int. Conf. on Formal Methods Europe, FME'02Copenhaguen, Denmark, 2002.
  - S. Colin, B. Legeard, and F. Peureux. **Preamble Computation in Automated Test Case Generation using Constraint Logic Programming**. The Journal of Software Testing, Verification and Reliability, 14(3):213--235, 2004.

- BZ-Testing-Tools
  - F. Ambert, F. Bouquet, S. Chemin, S. Guenaud, B. Legeard, F. Peureux, N. Vacelet, and M. Utting. **BZ-TT: A Tool-Set for Test Generation from Z and B using Constraint Logic Programming.** In Proc. of Formal Approaches to Testing of Software, 2002.

- Case studies
  - E. Bernard, B. Legeard, X. Luck, and F. Peureux. **Generation of test sequences from formal specifications: GSM 11-11 standard case study**. International Journal of Software Practice and Experience, 34(10):915--948, 2004.
  - Ph. Chevalley, B. Legeard, and J. Orsat. **Automated Test Case Generation for Space On-Board Software**. In Int. Conf. on Data Systems In Aerospace DASIA, 2005.

- Scenario-Based Testing
  - F. Dadeau and R. Tissot. **jSynoPSys -- A Scenario-Based Testing Tool based on the Symbolic Animation of B Machines**. *5th Int. Workshop on Model-Based Testing, MBT'2009)*, 2009.

- M. Utting and B. Legeard. **Practical Model-based Testing A tools approach**. Morgan & Kaufman



Thank you for your attention

Questions?