Model-Based Testing

From theory to practice (2/3)

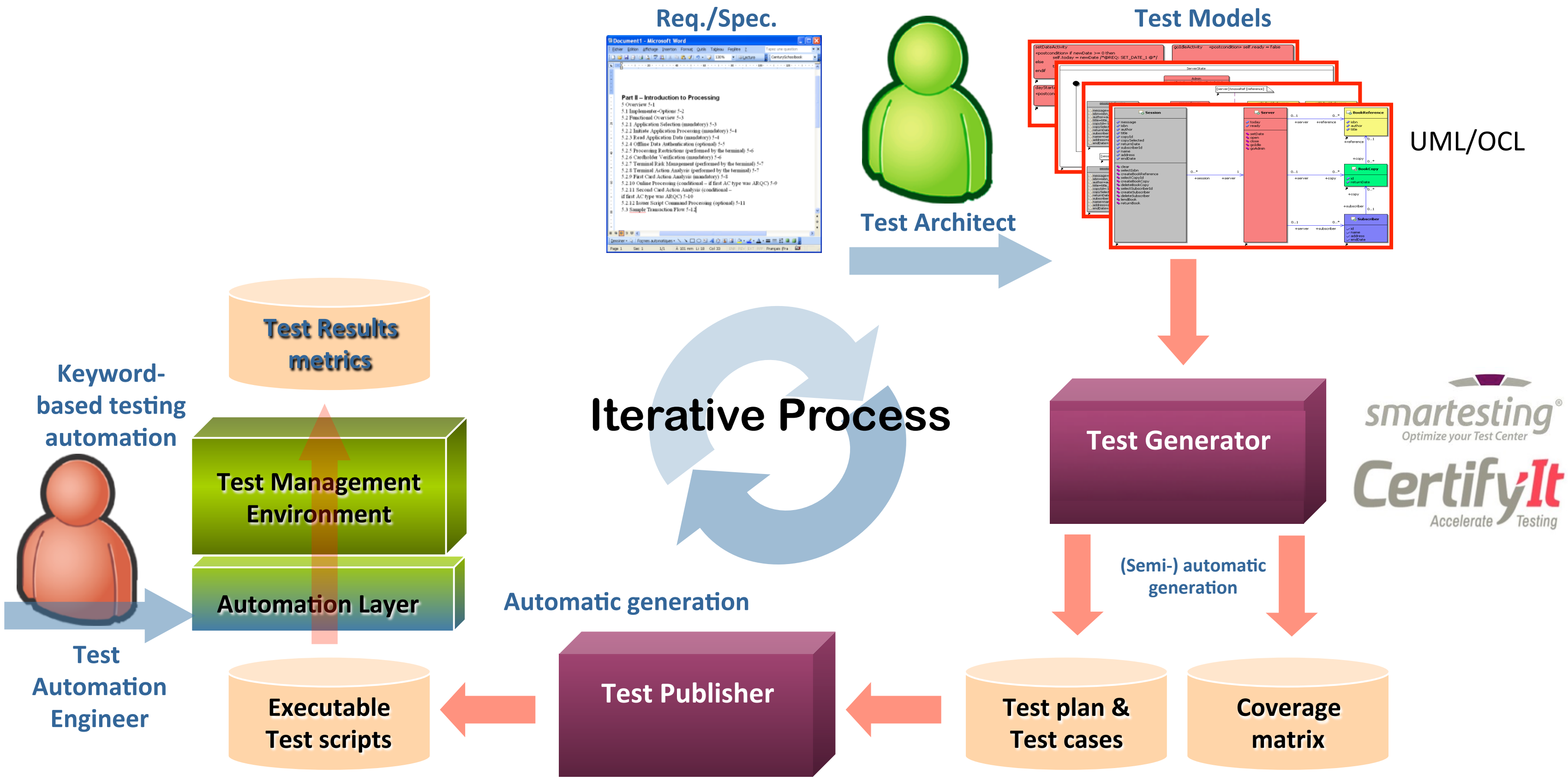
Bruno Legeard, **Frédéric Dadeau**, Elizabeta Fourneret

Ecole de Jeunes Chercheurs en Programmation – Nancy – June 24th, 2015

Agenda

1. Model-Based Testing with CertifyIt
2. Designing test models with UML/OCL
3. Test selection criteria & test generation processes
4. Concretization and conformance(s) relationship(s)
5. Summary: benefits/drawbacks of the MBT approaches

1. Model-Based Testing with CertifyIt



Req./Spec.

Document1 - Microsoft Word

Part II – Introduction to Processing

5 Overview 5-1
5.1 Implementer-Options 5-2
5.2 Functional Overview 5-3
5.2.1 Application Selection (mandatory) 5-3
5.2.2 Initiate Application Processing (mandatory) 5-4
5.2.3 Read Application Data (mandatory) 5-4
5.2.4 Offline Data Authentication (optional) 5-5
5.2.5 Processing Restrictions (performed by the terminal) 5-6
5.2.6 Cardholder Verification (mandatory) 5-6
5.2.7 Terminal Risk Management (performed by the terminal) 5-7
5.2.8 Terminal Action Analysis (performed by the terminal) 5-7
5.2.9 First Card Action Analysis (mandatory) 5-8
5.2.10 Online Processing (conditional – if first AC type was ARQC) 5-9
5.2.11 Second Card Action Analysis (conditional –
if first AC type was ARQC) 5-10
5.2.12 Issuer Script Command Processing (optional) 5-11
5.3 Sample Transaction Flow 5-12

Test Architect

Test Models

UML/OCL

Test Generator

(Semi-) automatic generation

Test plan & Test cases

Coverage matrix

Test Publisher

Automatic generation

Executable Test scripts

Automation Layer

Test Management Environment

Test Results metrics

Keyword-based testing automation

Test Automation Engineer

Iterative Process

2. Models: UML/OCL

- Unified Modeling Language + Object Constraint Language
  - use of a subset of UML, called UML4ST : no inheritance, binary associations, no dynamic creation of instances
  - adaptation of the usual semantics of OCL as an action language (to make OCL executable)

  → test model ≠ design model

- Three kinds of UML diagrams are considered:
  - class diagram → data model,
  - object diagram → initial state, and
  - statecharts → dynamics – not considered here, for simplicity

- Instead, OCL code is used to describe the behaviour of the operations

UML/OCL example

Running example (listen carefully, you'll work on it this afternoon!)

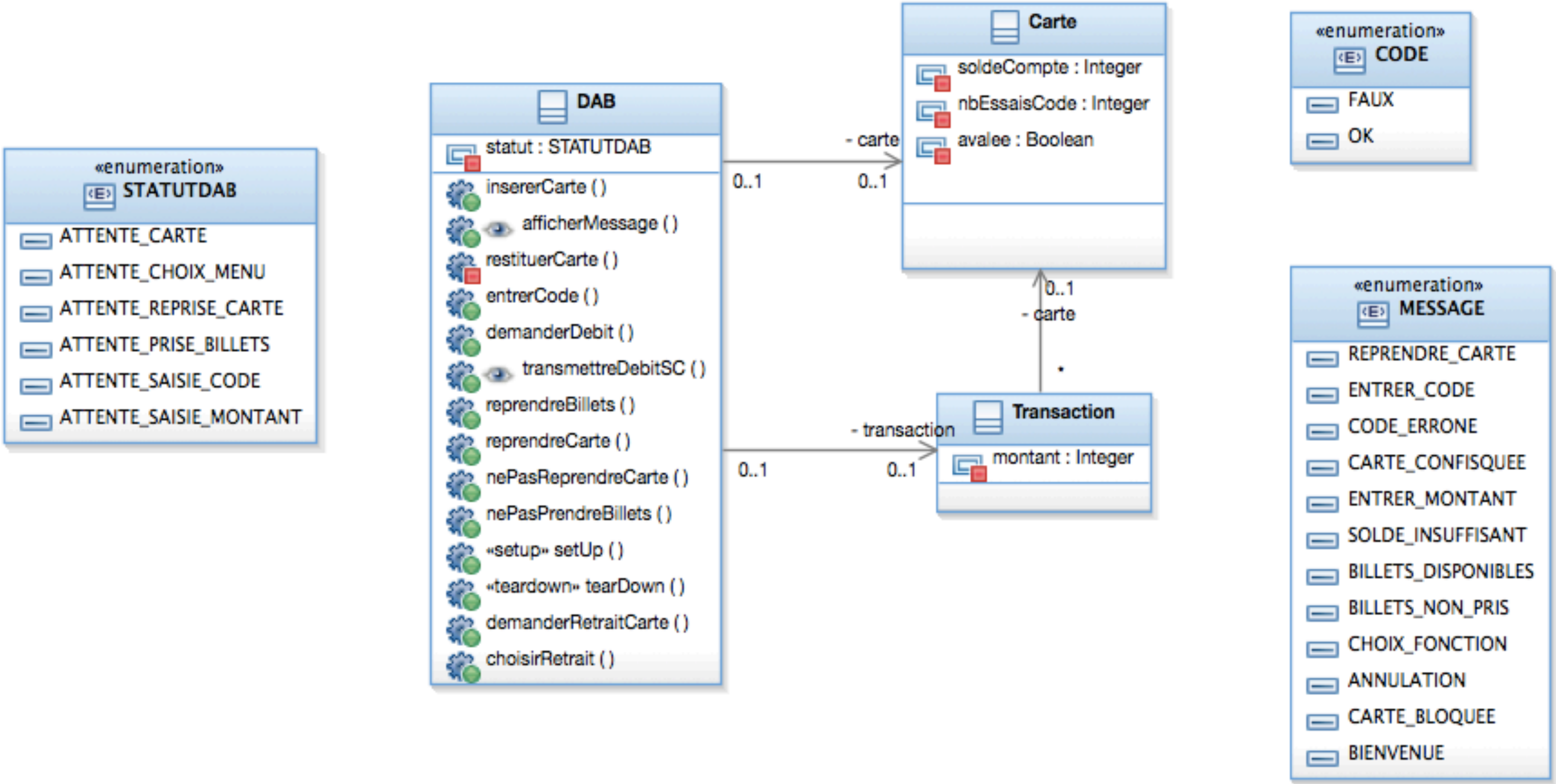
Automated Teller Machine (ATM) - withdraw cash with a credit card

- System Under Test (SUT) = cash machine
- Test data = credit cards with associated bank accounts
- Control points = reader, pad (0-9 + cancel, suppress, validate)
  → abstracted into « actions » (insert card, type PIN, etc.)
- Observation points = messages on the screen, card/bills ejected

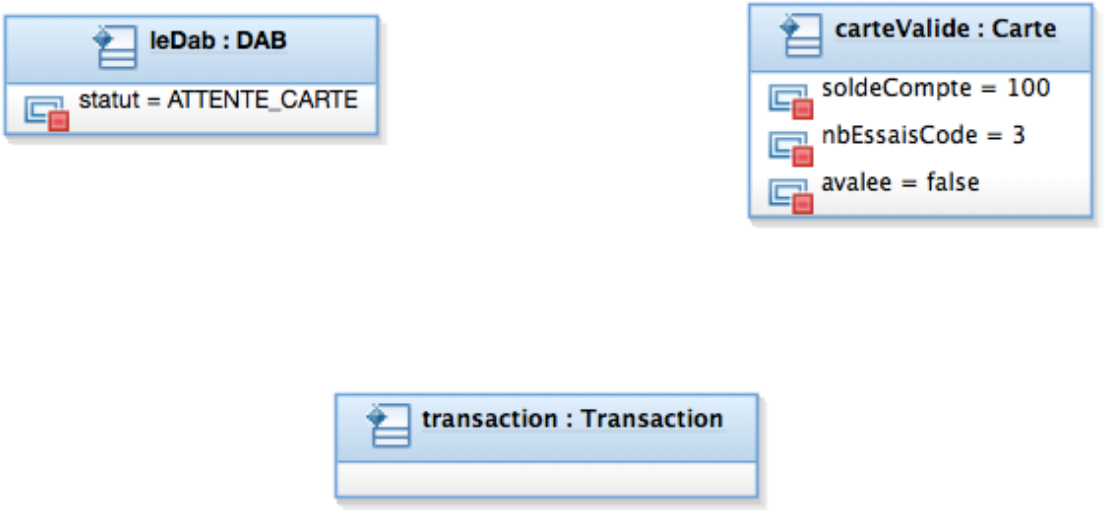
- Behaviours = « usual » behaviour of an ATM (functional testing)

No physical device → simulation on a web application (HTML5+JS)

Running example: ATM – Class diagram



Running example: ATM – Object diagram



leDab : DAB

statut = ATTENTE_CARTE

carteValide : Carte

soldeCompte = 100

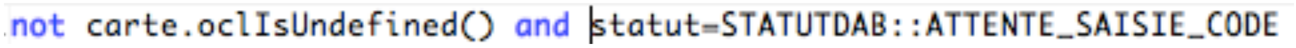
nbEssaisCode = 3

avalee = false

transaction : Transaction

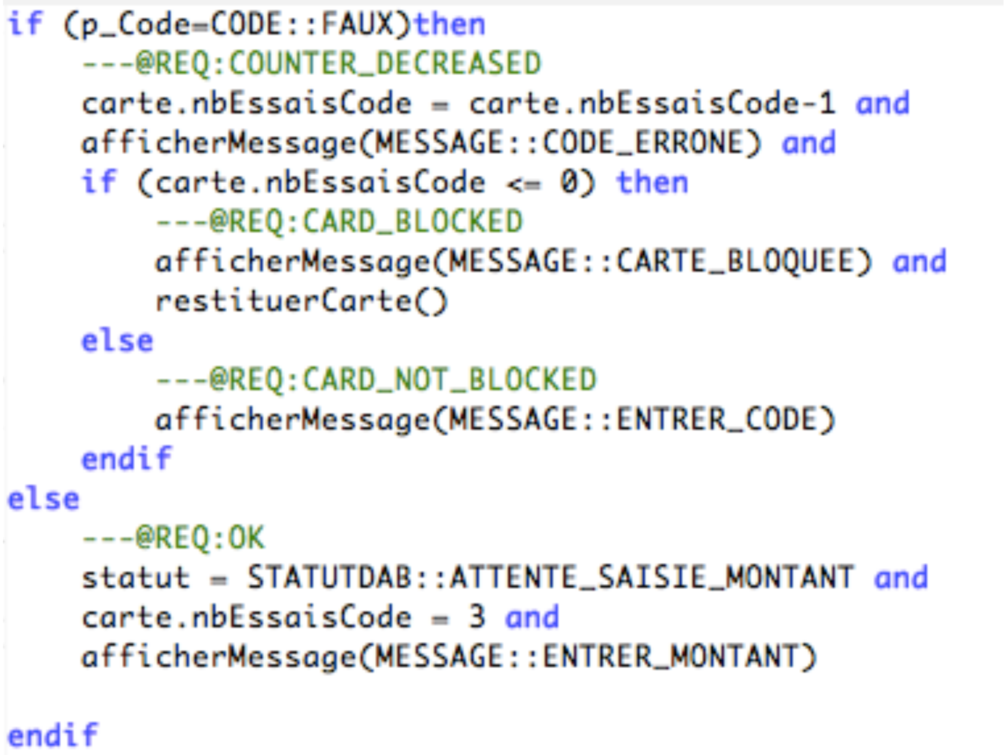
Running example: ATM – OCL

- Precondition of the entrerCode(PIN) operation



```
not carte.oclIsUndefined() and statut=STATUTDAB::ATTENTE_SAISIE_CODE
```

- Postcondition of the entrerCode(PIN) operation



```
if (p_Code=CODE::FAUX)then
    ---@REQ:COUNTER_DECREASED
    carte.nbEssaisCode = carte.nbEssaisCode-1 and
    afficherMessage(MESSAGE::CODE_ERRONE) and
    if (carte.nbEssaisCode <= 0) then
        ---@REQ:CARD_BLOCKED
        afficherMessage(MESSAGE::CARTE_BLOQUEE) and
        restituerCarte()
    else
        ---@REQ:CARD_NOT_BLOCKED
        afficherMessage(MESSAGE::ENTRER_CODE)
    endif
else
    ---@REQ:OK
    statut = STATUTDAB::ATTENTE_SAISIE_MONTANT and
    carte.nbEssaisCode = 3 and
    afficherMessage(MESSAGE::ENTRER_MONTANT)

endif
```

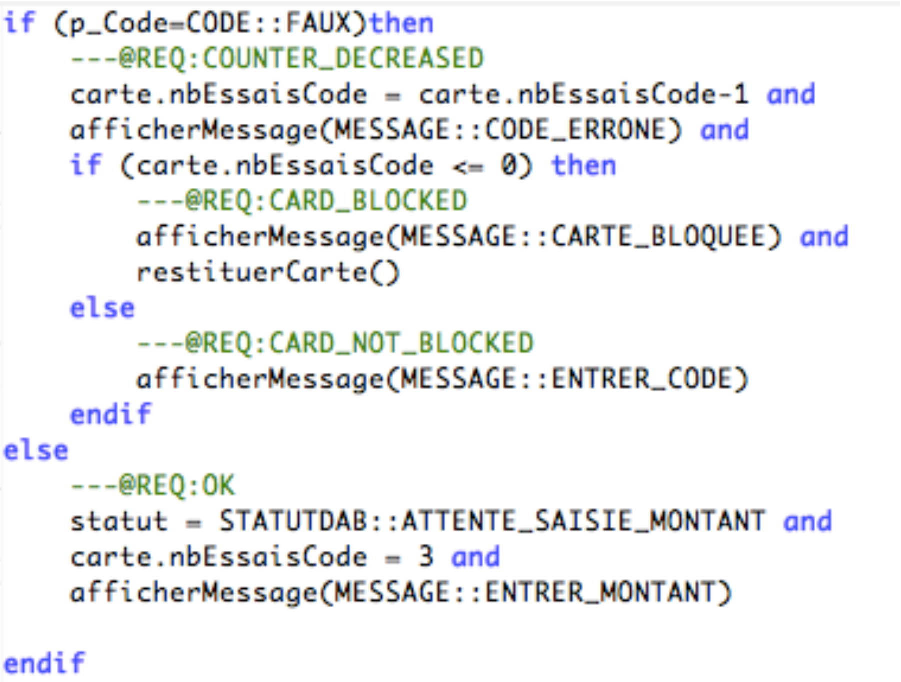
3. Test selection criteria & test generation

- Static test selection criteria
  - Structural coverage of the OCL code
  - Requirement coverage

- Dynamic test selection criteria
  - Test purposes (abstract test scenarios)
  - Temporal properties

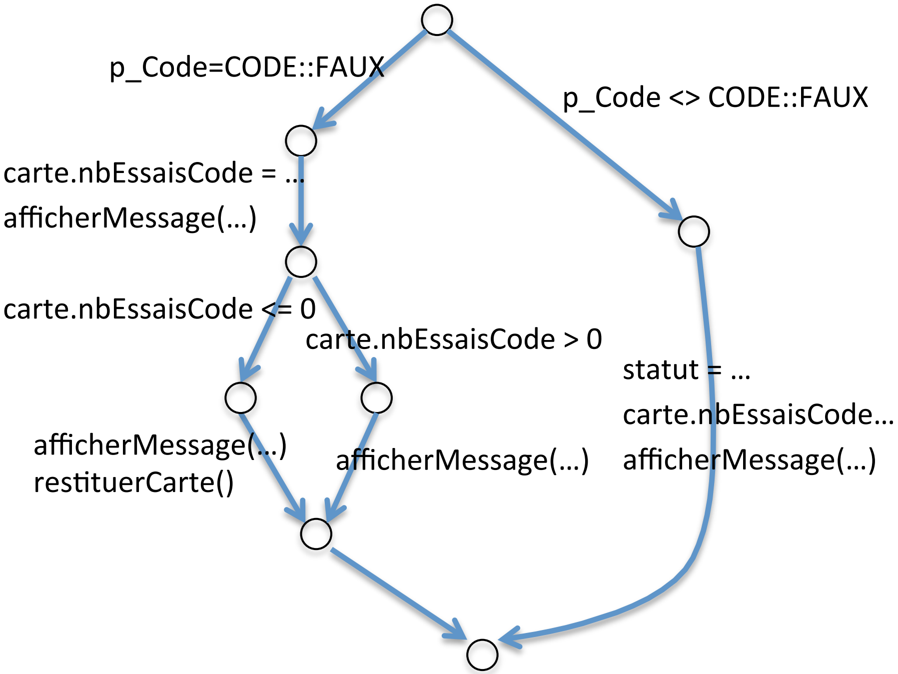
Static test selection criteria

- Goal: activate each behaviour of each operation of the SUT
  - Behaviour = branch in the CFG of the operation
  - Test Target = state that makes the execution of the behavior possible



```
if (p_Code=CODE::FAUX)then
    ---@REQ:COUNTER_DECREASED
    carte.nbEssaisCode = carte.nbEssaisCode-1 and
    afficherMessage(MESSAGE::CODE_ERRONE) and
    if (carte.nbEssaisCode <= 0) then
        ---@REQ:CARD_BLOCKED
        afficherMessage(MESSAGE::CARTE_BLOQUEE) and
        restituerCarte()
    else
        ---@REQ:CARD_NOT_BLOCKED
        afficherMessage(MESSAGE::ENTRER_CODE)
    endif
else
    ---@REQ:OK
    statut = STATUTDAB::ATTENTE_SAISIE_MONTANT and
    carte.nbEssaisCode = 3 and
    afficherMessage(MESSAGE::ENTRER_MONTANT)

endif
```



p_Code=CODE::FAUX

p_Code <> CODE::FAUX

carte.nbEssaisCode = …

afficherMessage(…)

carte.nbEssaisCode <= 0

carte.nbEssaisCode > 0

statut = …

carte.nbEssaisCode…

afficherMessage(…)

afficherMessage(…)
restituerCarte()

afficherMessage(…)

Static test selection criteria

- For each test target, automatically explore the model states and compute a sequence of operations that reaches the target

- Shape of a test case:
  - <preamble> = sequence of operations, from the initial state that reaches the target
  - <body> = invocation of the operation to active the targeted behavior
  - <observation> = possible additional operations that can be executed to check that the targeted operation was correctly executed

- Test cases can be merged to minimize the size of the test suite

Static test selection criteria

Examples of functional tests:

-
  ```
  leDab.insererCarte(carteValide)         // @REQ: OK
  leDab.choisirRetrait()                  // @REQ: OK
  leDab.entrerCode(OK)                    // @REQ: OK
  leDab.demanderDebit(50)                 // @REQ: OK
  leDab.reprendreCarte()                  // @REQ: OK, @REQ: TRANSACTION_DONE
  leDab.reprendreBillets()                // @REQ: OK
  ```

-
  ```
  leDab.insererCarte(carteValide)         //@REQ: OK
  leDab.choisirRetrait()                  // @REQ: OK
  leDab.entrerCode(OK)                    // @REQ: OK
  leDab.demanderDebit(150)                // @REQ: INSUFFICIENT_BALANCE
  ```

Static test selection criteria - limitations

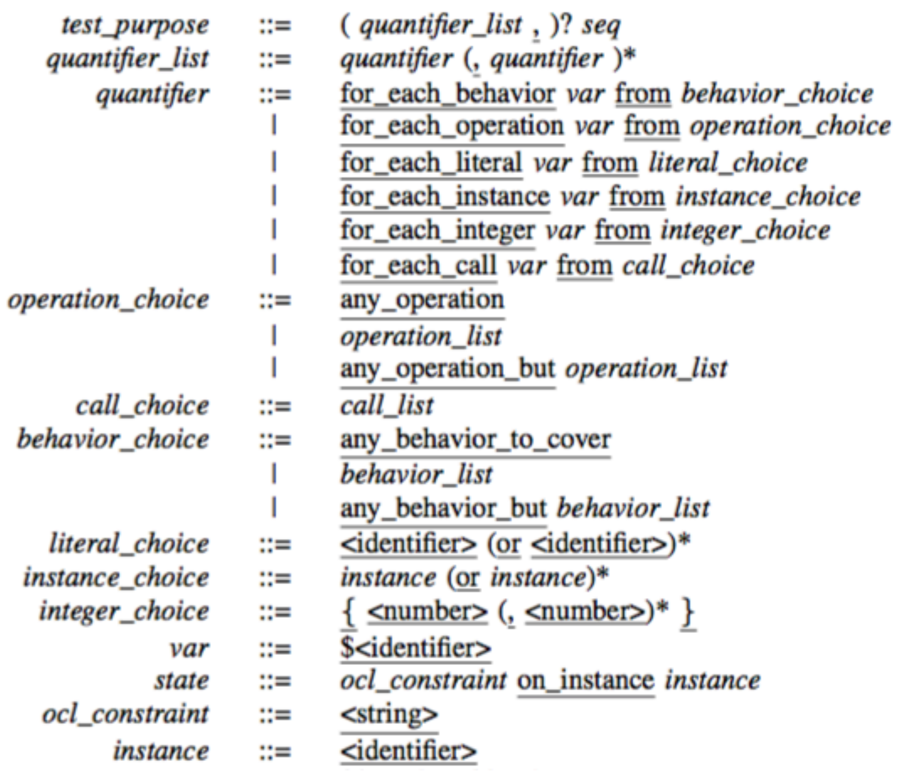
- Limitations of automated testing based on static criteria (structural/ requirement coverage)
  - test cases with limited size (steps)
  - difficulty to take into account the dynamics of the system (must be hard-coded into the model)
  - possible issues with the test target's reachability

- Two complementary ways to drive the test generation:
  - test scenarios
  - temporal test properties

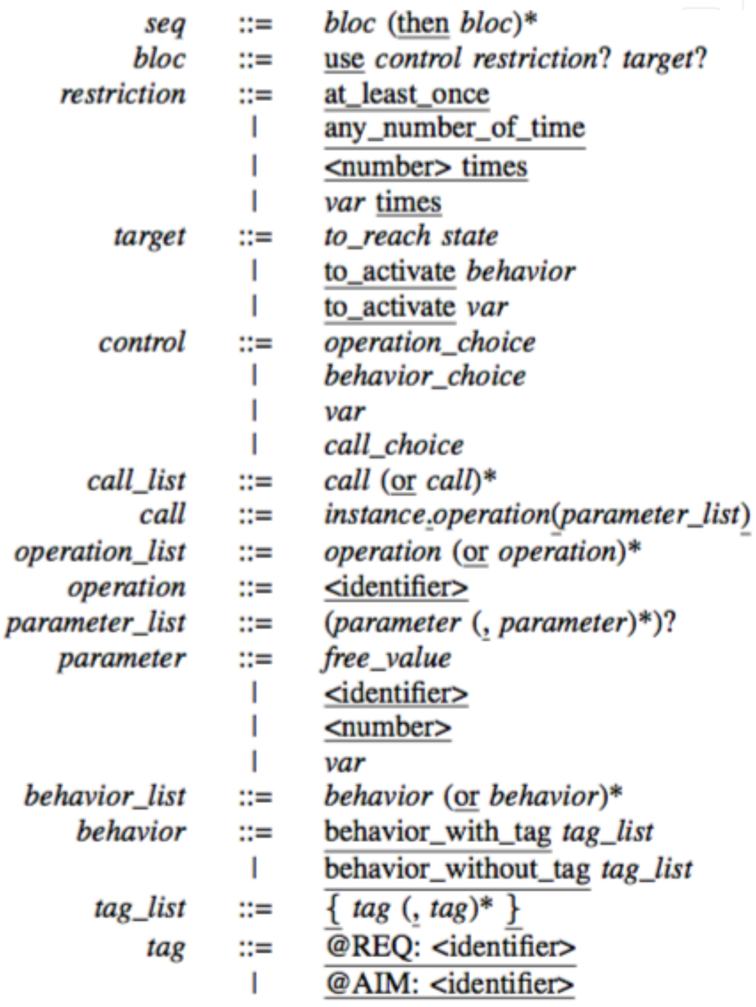
Dynamic criteria: test purposes

- Test scenarios that help the user describing test sequences that cannot be computed by the tool
- Based on regular expressions involving operations and state predicates
- However, textual description, close to natural language (to help the test designer)
- Unfolded on the model to be instantiated as a test case

Test purposes syntax



```
    test_purpose  ::=  ( quantifier_list , )? seq
 quantifier_list  ::=  quantifier (, quantifier )*
      quantifier  ::=  for_each_behavior var from behavior_choice
                    |  for_each_operation var from operation_choice
                    |  for_each_literal var from literal_choice
                    |  for_each_instance var from instance_choice
                    |  for_each_integer var from integer_choice
                    |  for_each_call var from call_choice
operation_choice  ::=  any_operation
                    |  operation_list
                    |  any_operation_but operation_list
     call_choice  ::=  call_list
 behavior_choice  ::=  any_behavior_to_cover
                    |  behavior_list
                    |  any_behavior_but behavior_list
  literal_choice  ::=  <identifier> (or <identifier>)*
 instance_choice  ::=  instance (or instance)*
  integer_choice  ::=  { <number> (, <number>)* }
             var  ::=  $<identifier>
           state  ::=  ocl_constraint on_instance instance
  ocl_constraint  ::=  <string>
        instance  ::=  <identifier>
```

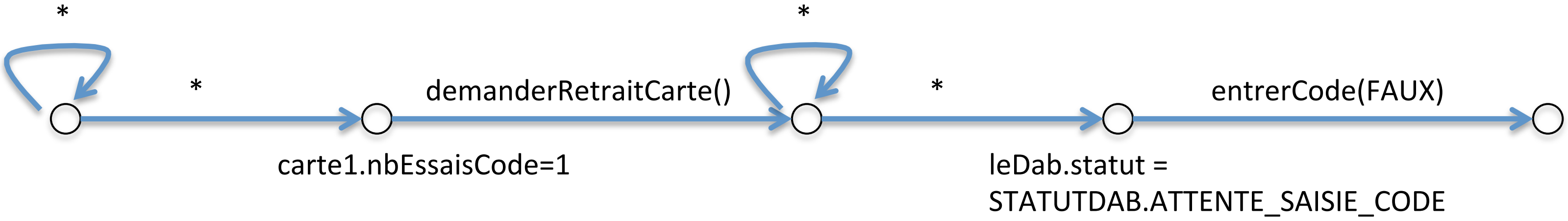


```
           seq  ::=  bloc (then bloc)*
          bloc  ::=  use control restriction? target?
   restriction  ::=  at_least_once
                  |  any_number_of_time
                  |  <number> times
                  |  var times
        target  ::=  to_reach state
                  |  to_activate behavior
                  |  to_activate var
       control  ::=  operation_choice
                  |  behavior_choice
                  |  var
                  |  call_choice
     call_list  ::=  call (or call)*
          call  ::=  instance.operation(parameter_list)
operation_list  ::=  operation (or operation)*
     operation  ::=  <identifier>
parameter_list  ::=  (parameter (, parameter)*)?
     parameter  ::=  free_value
                  |  <identifier>
                  |  <number>
                  |  var
 behavior_list  ::=  behavior (or behavior)*
      behavior  ::=  behavior_with_tag tag_list
                  |  behavior_without_tag tag_list
      tag_list  ::=  { tag (, tag)* }
           tag  ::=  @REQ: <identifier>
                  |  @AIM: <identifier>
```

Test purposes

Example on the ATM: a test scenario that checks that pin retry counter is correctly implemented

```
use any_operation any_number_of_times
          to_reach "nbEssaisCode = 1" on_instance carte1
then use leDab.demanderRetraitCarte()
then use any_operation any_number_of_times
          to_reach "statut=STATUTDAB::ATTENTE_SAISIE_CODE" on_instance leDab
then use leDab.entrerCode(FAUX)                  // should block and eject the card
```



Test purposes

Example on the ATM: a test scenario that checks that pin retry counter is correctly implemented

Once unfolded on the model:

```
leDab.insererCarte(carteValide)
leDab.choisirRetrait()
leDab.entrerCode(FAUX)
leDab.entrerCode(FAUX)
leDab.demanderRetraitCarte()
leDab.reprendreCarte()
leDab.insererCarte(carteValide)
leDab.choisirRetrait()
leDab.entrerCode(FAUX)
```

Dynamic criteria: test properties

- Some test scenarios address specific test intentions, that could be formalized by high-level properties

- TOCL = Temporal OCL
  - overlay of OCL to express temporal properties
  - based on Dwyer *et al.* property patterns [DAC99]
  - does not require the use of a complex formalism (e.g. LTL, CTL)

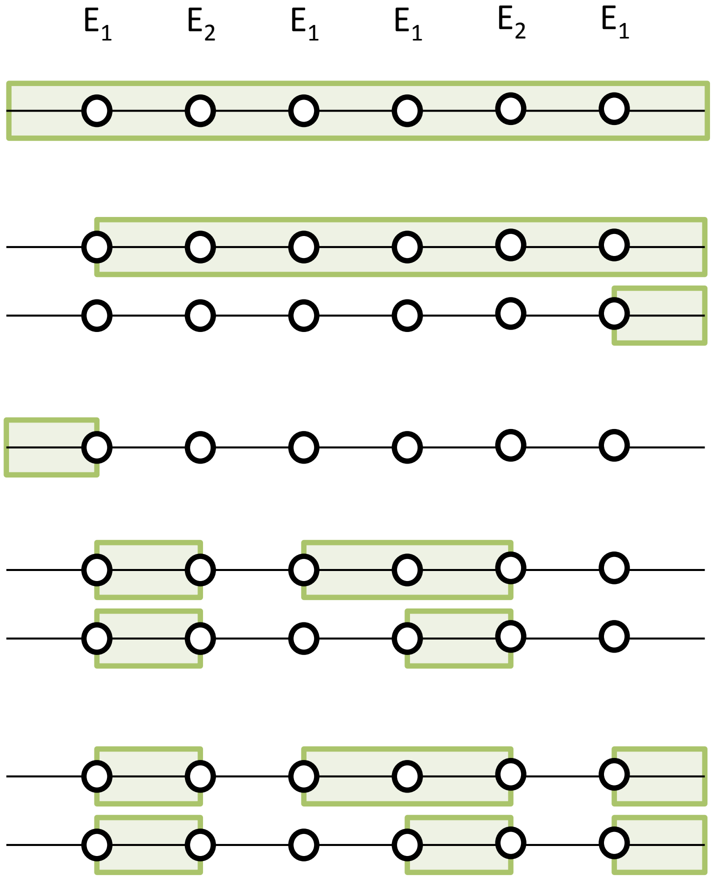
- Property = Pattern + Scope
  - Pattern: describes occurrences or orderings of events
  - Scope: describes the observation window on which the pattern is supposed to hold

[DAC99] M. Dwyer, G. Avrunin, and J. Corbett. *Patterns in property specifications for finite-state verification*. ICSE'99.

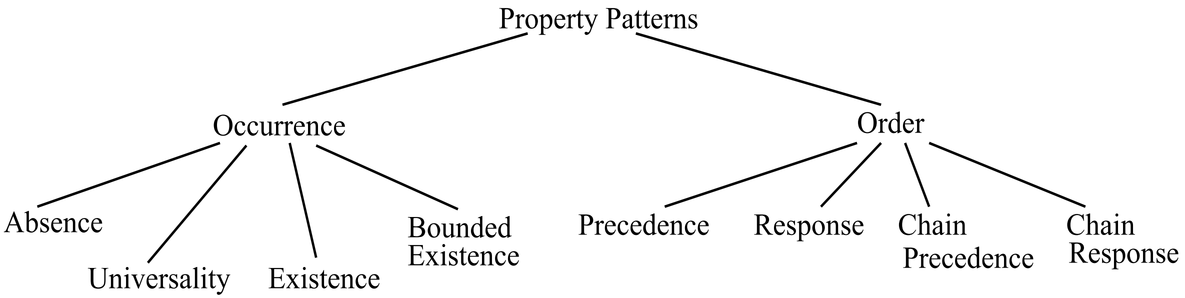
Temporal Properties in TOCL

Scopes

- globally
- after E_1
  - after last E_1
- before E_1
- between E_1 and E_2
  - between last E_1 and E_2
- after E_1 until E_2
  - after last E_1 until E_2



Temporal Properties in TOCL

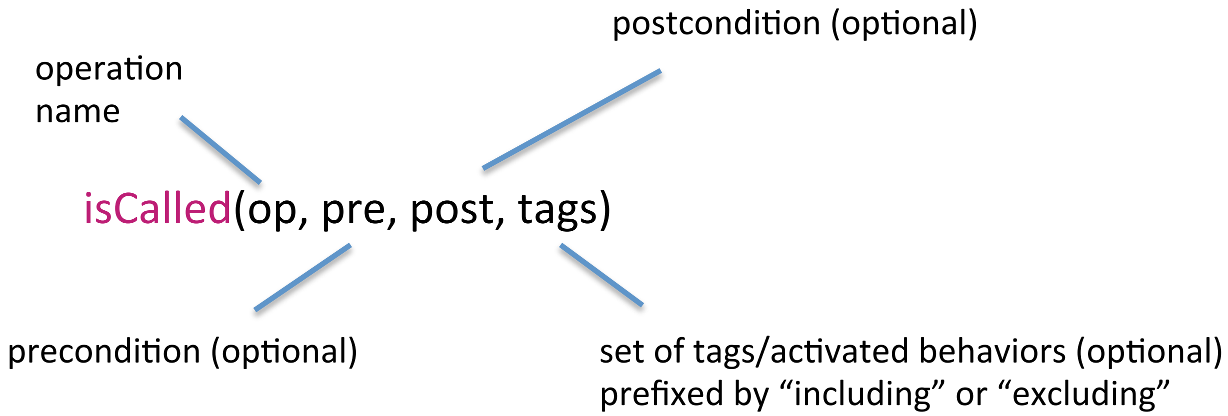


Patterns

- always P
- never E
- eventually E at least/at most/exactly k times
- E_1 [directly] precedes E_2
- E_1 [directly] follows E_2

Temporal Properties in TOCL

Events: operation calls



postcondition (optional)

operation
name

isCalled(op, pre, post, tags)

precondition (optional)

set of tags/activated behaviors (optional)
prefixed by “including” or “excluding”

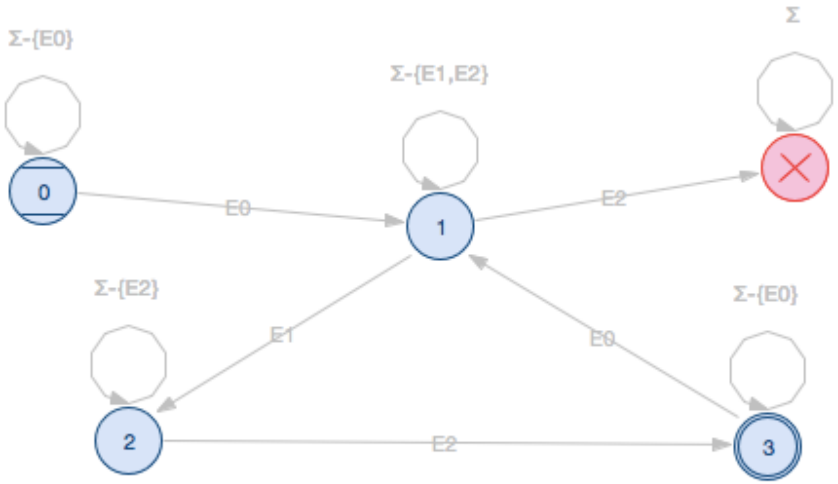
Temporal Properties in TOCL

"Once a card is inserted, it is necessary to authenticate to get bills."

between isCalled(leDab.insererCarte,including:{@REQ:OK})
and isCalled(leDab.reprendreBillets,including:{@REQ:OK})

eventually isCalled(leDab.entrerCode,including:{@REQ:OK})
at least 1 times

E0 = insererCarte
E1 = entrerCode
E2 = reprendreBillets

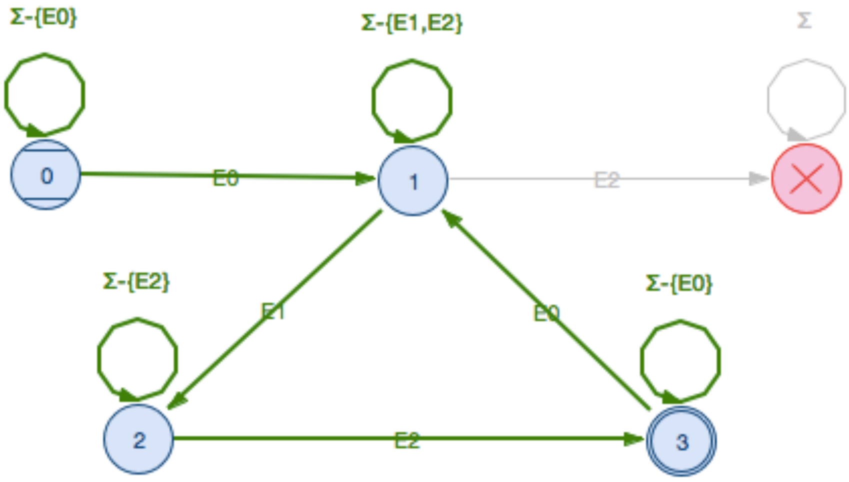


Using the properties for testing

- Two possible uses for these properties

1. Measure the quality of a test suite

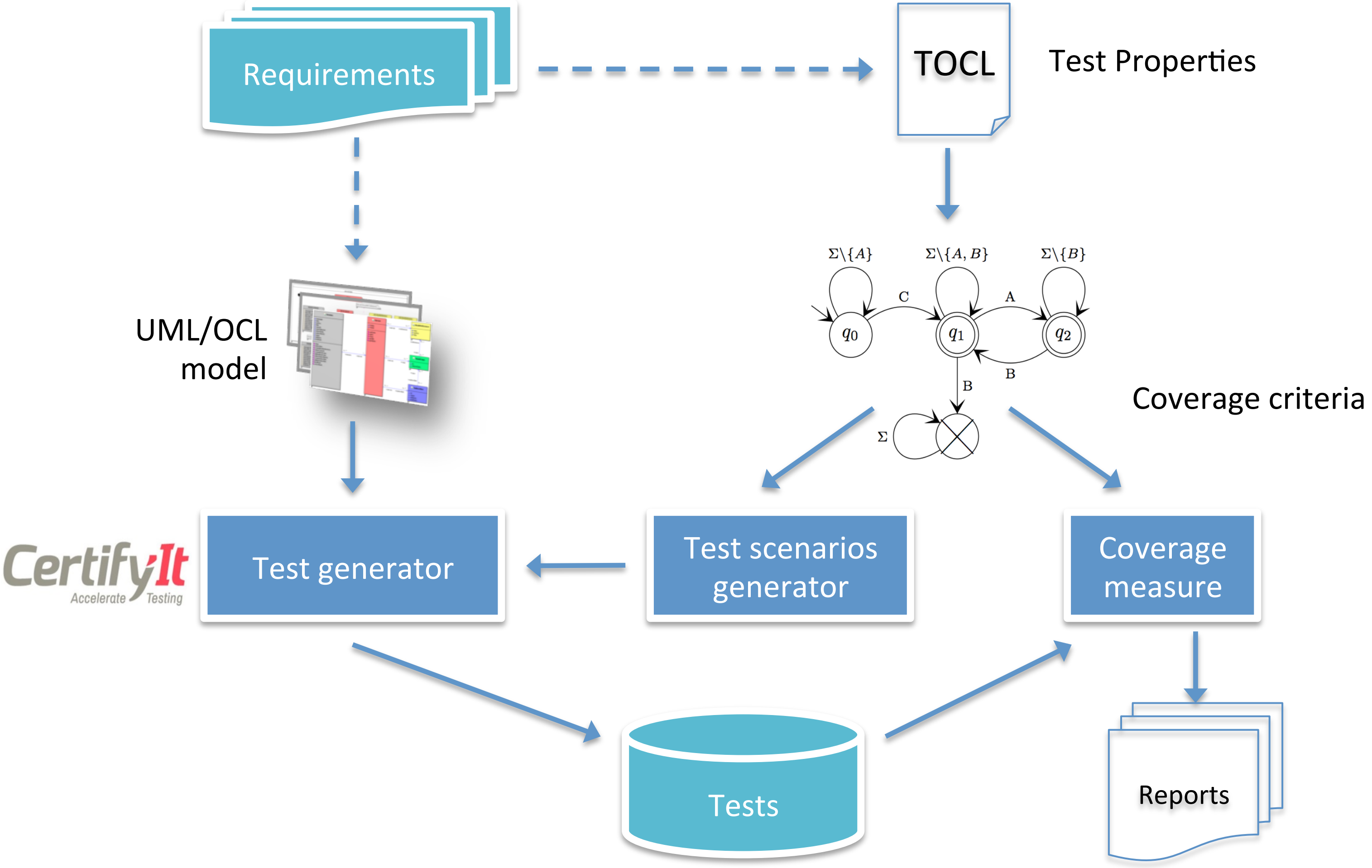
leDab.insererCarte(carteValide)	0 → 1
leDab.choisirRetrait()	1 → 1
leDab.entrerCode(OK)	1 → 2
leDab.demanderDebit(50)	2 → 2
leDab.reprendreCarte()	2 → 2
leDab.reprendreBillets()	2 → 3



2. Generate new tests

use any_operation any_number_of_times then
use leDab.insererCarteValide to_activate {@REQ:OK} then
use any_operation any_number_of_times then
use entrerCode() to_activate {@REQ:OK} then
use any_operation any_number_of_times then
use reprendreBillets() to_activate {@REQ:OK} … x2

Use of temporal test properties



Requirements

TOCL

Test Properties

UML/OCL model

Coverage criteria

Test generator

Test scenarios generator

Coverage measure

Tests

Reports

Interest of test properties

- Language is easy to learn and use to design test properties
- Usefulness of the coverage reports
  - shows which part of the properties are not covered by the tests
- Relevance of the coverage criteria
  - Property automata are rarely 100% covered by the functional test suite
  - “Shows test configurations that one may not easily think of”
- Unintended use of the properties: model validation
  - Use of the test cases coverage measure to detect violations of the property by the model

4. Concretization and conformance

- Two issues to consider:
  - Bridge the gap between the abstract and concrete level
    - Control : abstract operations + parameters
    - Observations : return values, specific operations
  - Implement the conformance relationship and establish the test verdict:
    - « Pass »
    - « Fail »
    - possibly, « Inconclusive »

Concretization

- abstract tests → concrete tests
- Control points:
  - Map abstract operations with concrete « actions »
  - Also map parameters list (if necessary adapt it)
  - Translate abstract values into concrete ones (especially enumerations)
- Observations:
  - Return values (to be translated) of the operations
  - Dedicated operations (stereotyped « observation » in the model)

→ of the utmost importance: determines the accuracy of the test

→ hopefully, the model provides the test oracle (the expected result)

Conformance relationship

Many different conformance relationships: isomorphism, bisimulation, trace equivalence, etc.

Reasonable compromise: ioco [Tretmans’96] defined on IOLTS

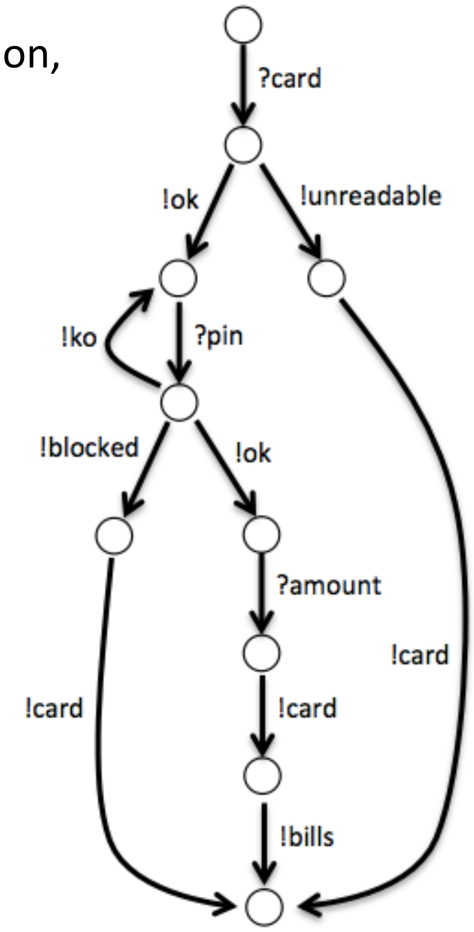
IOLTS = $< Q, A, \rightarrow, q0 >$

- Q = set of states
- $A = A_i \cup A_o \cup \{\tau\}$ with
  - A_i = input actions (prefixed by ?)
  - A_o = output actions (prefixed by !)
  - τ = internal action
- $\rightarrow \subseteq Q \times A \times Q$
- q0 = initial state

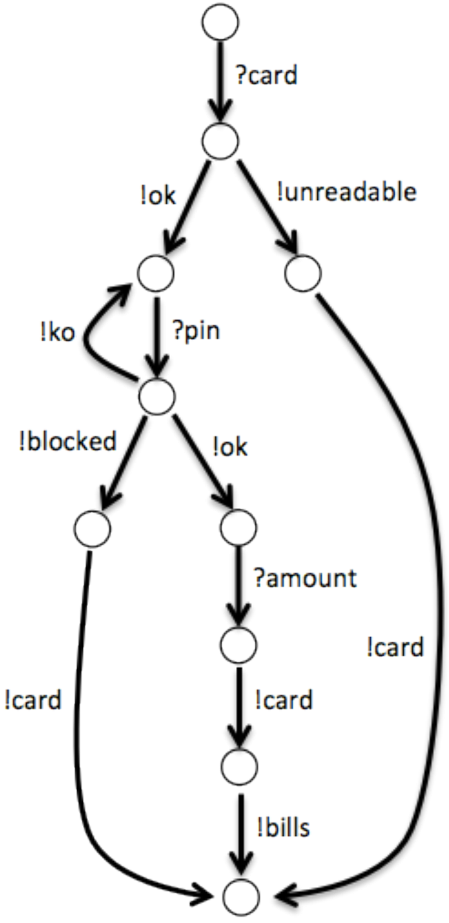
δ : quiescence (observation of no output) : deadlock/livelock

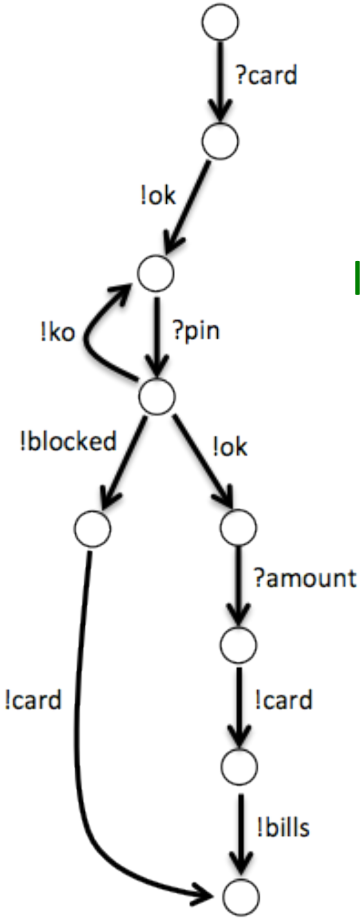
IUT ioco S: $\forall \sigma$: Straces(S) : out(IUT after σ) $\subseteq$ out(S after σ)

After each suspended trace (ie. an execution up to a quiescence), IUT exhibits only outputs and quiescences present in S.



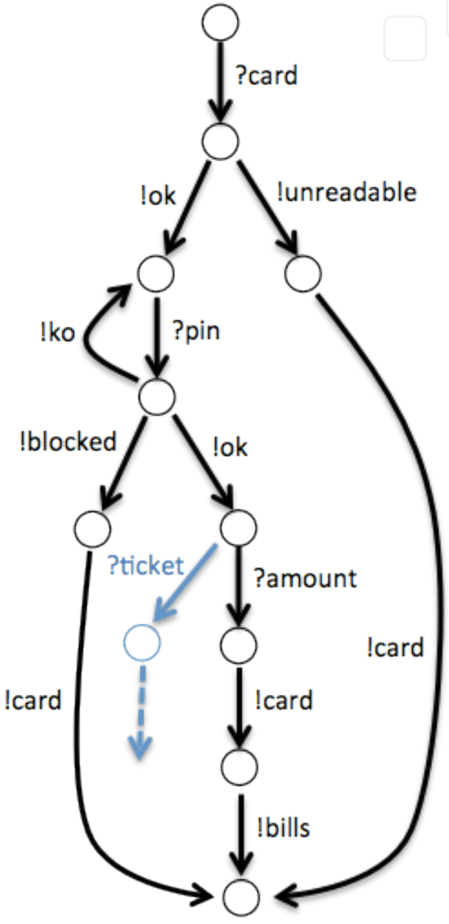
Conformance relationship





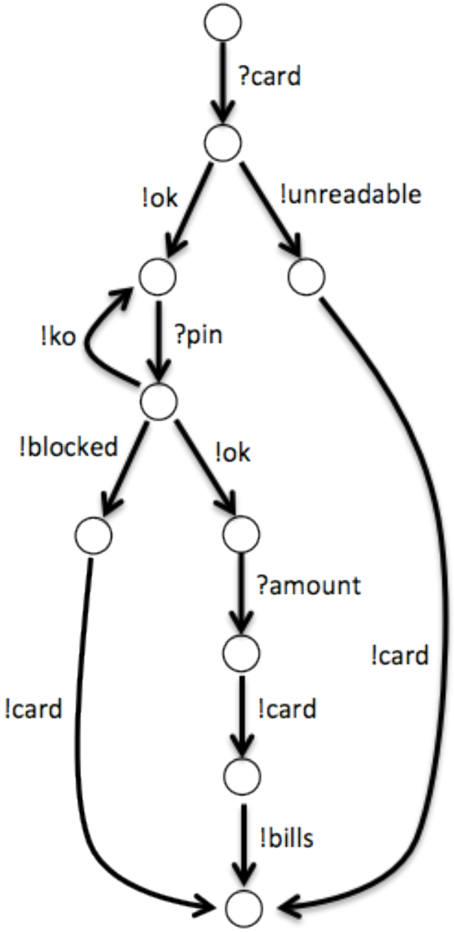
Implementation choice

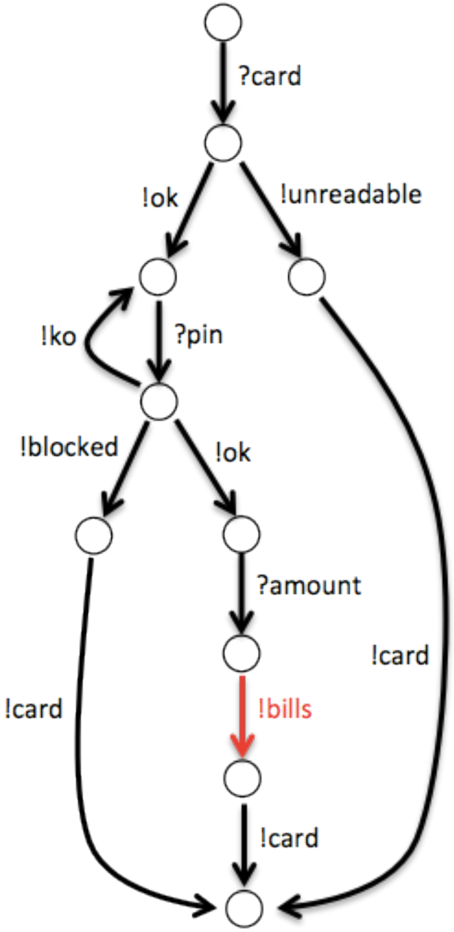
IOCO



Implementation of partial specification

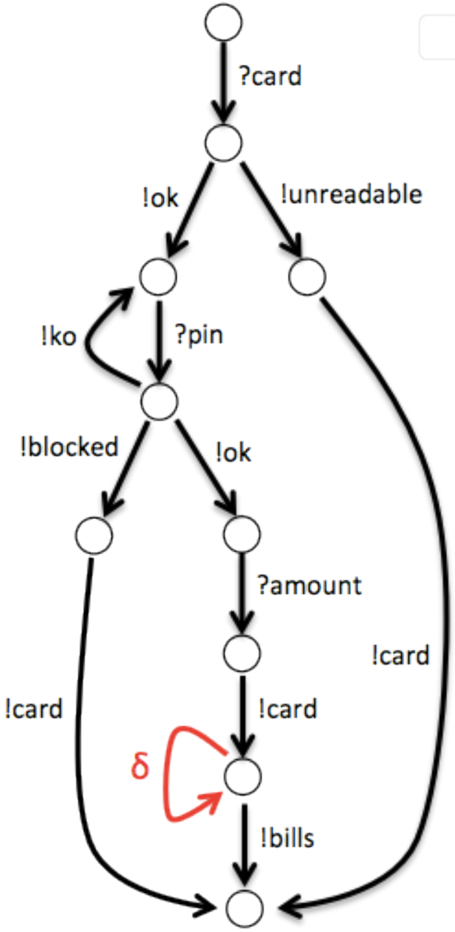
Conformance relationship





Unexpected output

~~IOCO~~



Unexpected quiescence

5. Conclusion: benefits and drawbacks

Benefits:

- back to back validation of a system: a comparison of two point of views
- functional testing: does not aim at runtime errors (null pointers, divisions by 0, etc.) but focus on specification mistakes (40% of the errors in a program)
- look for automation

Drawbacks:

- Model design step:
  - learning curve to take into account (language)
  - keep in mind you design a test model, not a design model
- Test generator: need to know how it works to produce the right tests
- Test verdict:
  - implement the conformance relationship you want (ioco might not be sufficient!)
  - in case of non-conformance: where is the error?

Our advise: perform MBT iteratively and incrementally

Some references

- Mark Utting, Alexander Pretschner, and Bruno Legeard. A taxonomy of model-based testing approaches. Software Testing, Verification and Reliability, 22(5):297--312, 2012.
- Julien Botella, Jürgen Grossmann, Bruno Legeard, Fabien Peureux, Martin Schneider, and Fredrik Seehusen. Model-Based Security Testing with Test Patterns. In UCAAT 2014, 2nd User Conference on Advanced Automated Testing, Munich, Germany, September 2014. ETSI.
- Frédéric Dadeau, Kalou Cabrera Castillos, and Jacques Julliand. Coverage Criteria for Model-Based Testing using Property Patterns. In A.K. Petrenko and H. Schlingloff, editors, MBT 2014, 9th Workshop on Model-Based Testing, Satellite workshop of ETAPS 2014, volume 141 of EPTCS, Grenoble, France, pages 29--43, April 2014. Open Publishing Association.
- Julien Botella, Fabrice Bouquet, Jean-François Capuron, Franck Lebeau, Bruno Legeard, and Florence Schadle. Model-Based Testing of Cryptographic Components -- Lessons Learned from Experience. In ICST'13, 6th IEEE Int. Conf. on Software Testing, Verification and Validation, pages 192--201, March 2013.
- Kalou Cabrera Castillos, Frédéric Dadeau, and Jacques Julliand. Scenario-Based Testing from UML/OCL Behavioral Models -- Application to POSIX Compliance. STTT, International Journal on Software Tools for Technology Transfer, 13(5):431--448, 2011. Note: Special Issue on Verified Software: Tools, Theory and Experiments (VSTTE'09)
- Fabrice Bouquet, Christophe Grandpierre, Bruno Legeard, and Fabien Peureux. A test generation solution to automate software testing. In AST'08, 3rd Int. workshop on Automation of Software Test, Leipzig, Germany, pages 45--48, May 2008. ACM Press.