Coverage Criteria for Model-Based Testing using Property Patterns

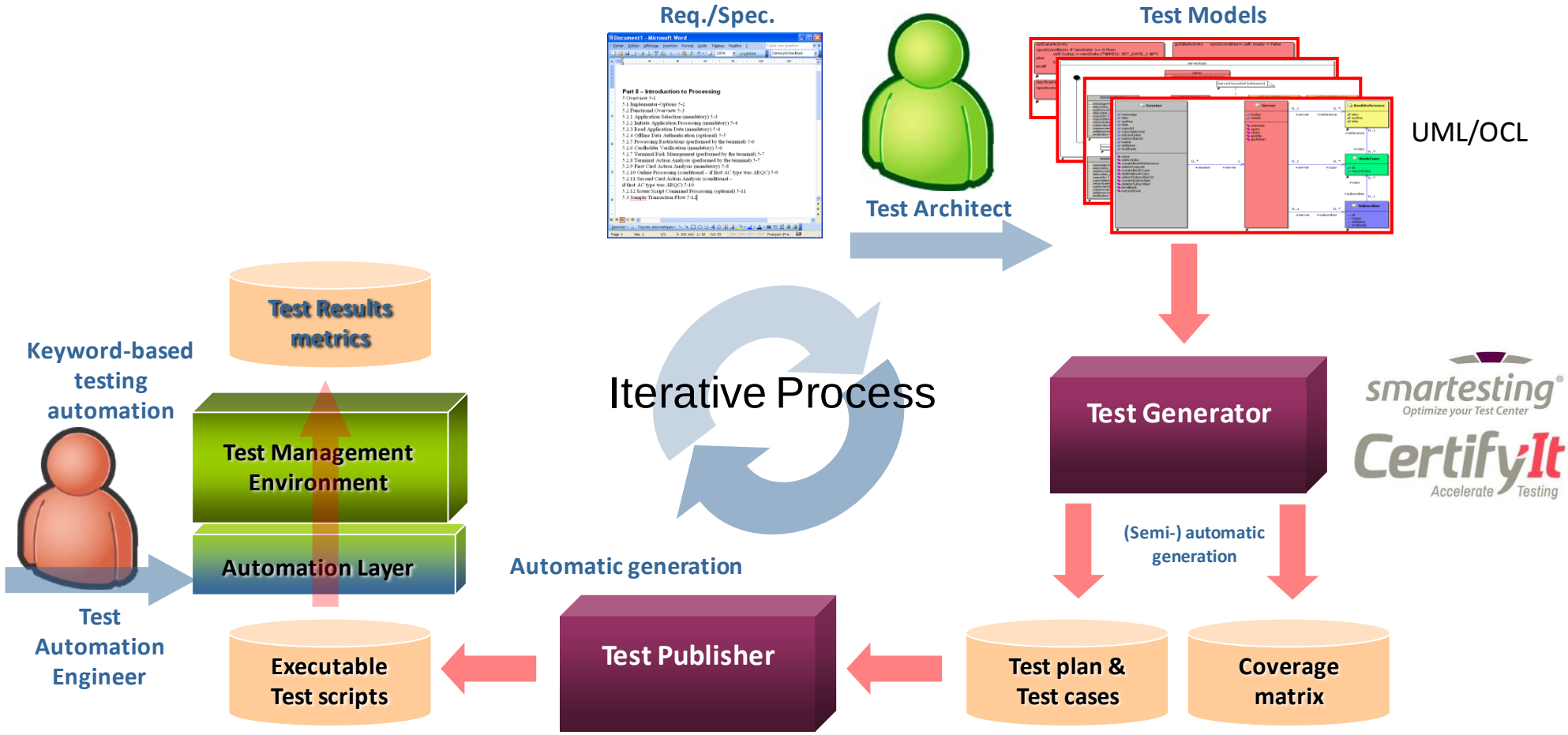
Kalou Cabrera Castillos[1], **Frédéric Dadeau**[2], Jacques Julliand[2]

[1] LAAS – Toulouse, France

[2] FEMTO-ST – Besançon, France

MBT workshop – April 6th, 2014

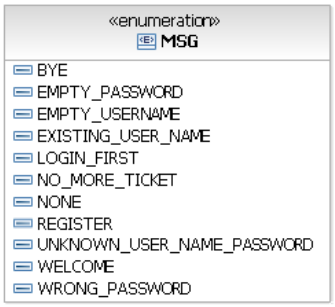
Context: Model-Based Testing

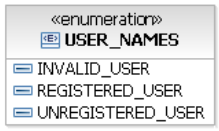


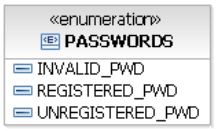
Context: Smartesting CertifyIt and UML4ST

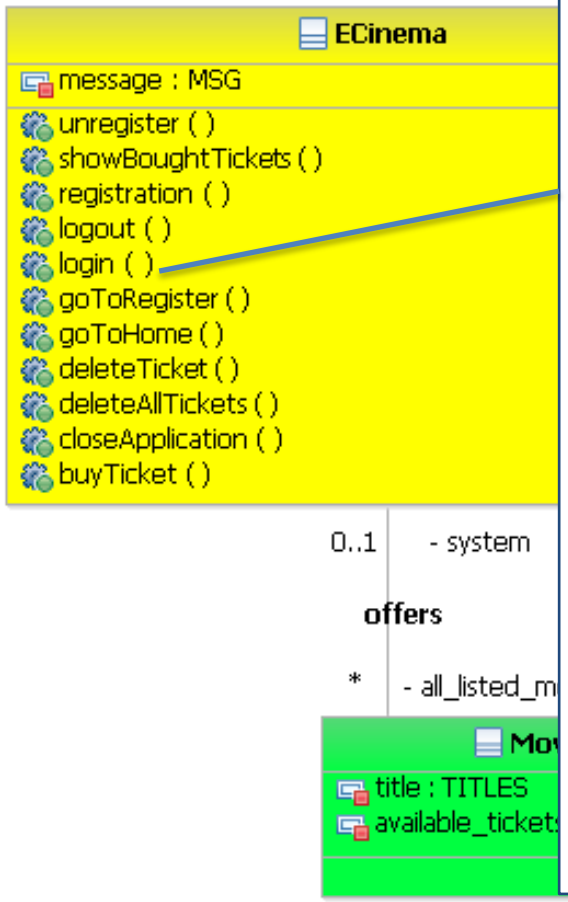
- Functional test generation from UML/OCL models
  - Use of a subset of UML, called UML4ST
  - 3 UML diagrams: class diagrams (data model), object (initial state), and statecharts (dynamics)
  - OCL code is used to describe the behaviour of the operations

Running example: eCinema










```
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
```

Context: Smartesting CertifyIt and UML4ST

- Functional test generation from UML/OCL models
  - Use of a subset of UML, called UML4ST
  - 3 UML diagrams: class diagrams (data model), object (initial state), and statecharts (dynamics)
  - OCL code is used to describe the behaviour of the operations

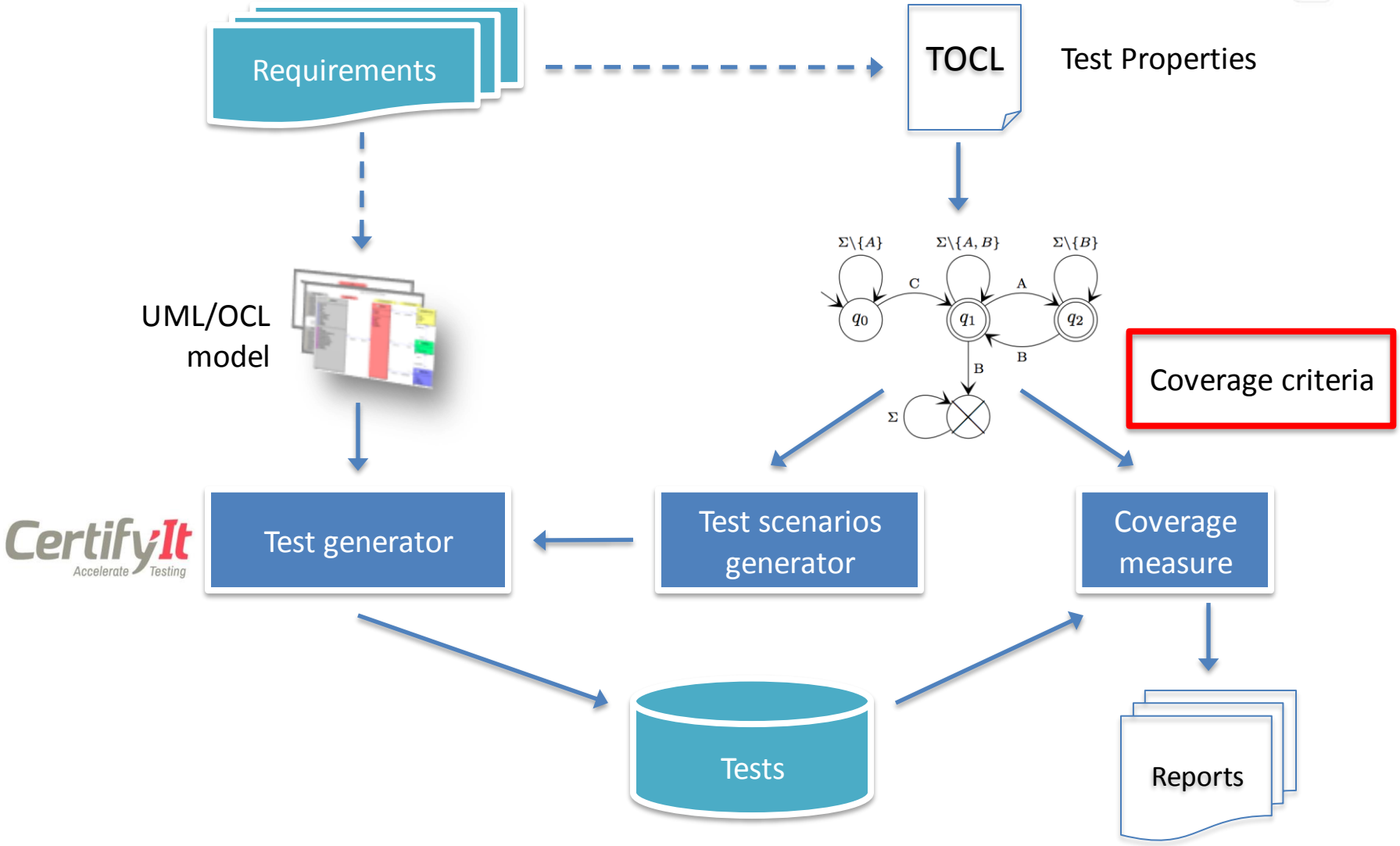
- How Smartesting CertifyIt works
  - aims at covering of the behaviours of the operations (OCL code coverage)
  - retrieves the traceability requirements (annotations in the code) covered by the tests

Motivations

- Limitations of automated testing based on 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

- Our proposal: use temporal test properties
  - How to express the test properties easily?
  - How to characterize relevant tests?

Summary of the approach



Outline

- Context and motivations
- Property pattern language
- Coverage criteria: nominal and robustness
- Experimental results
- Conclusion and perspectives

Design of Temporal Properties using TOCL

- 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)

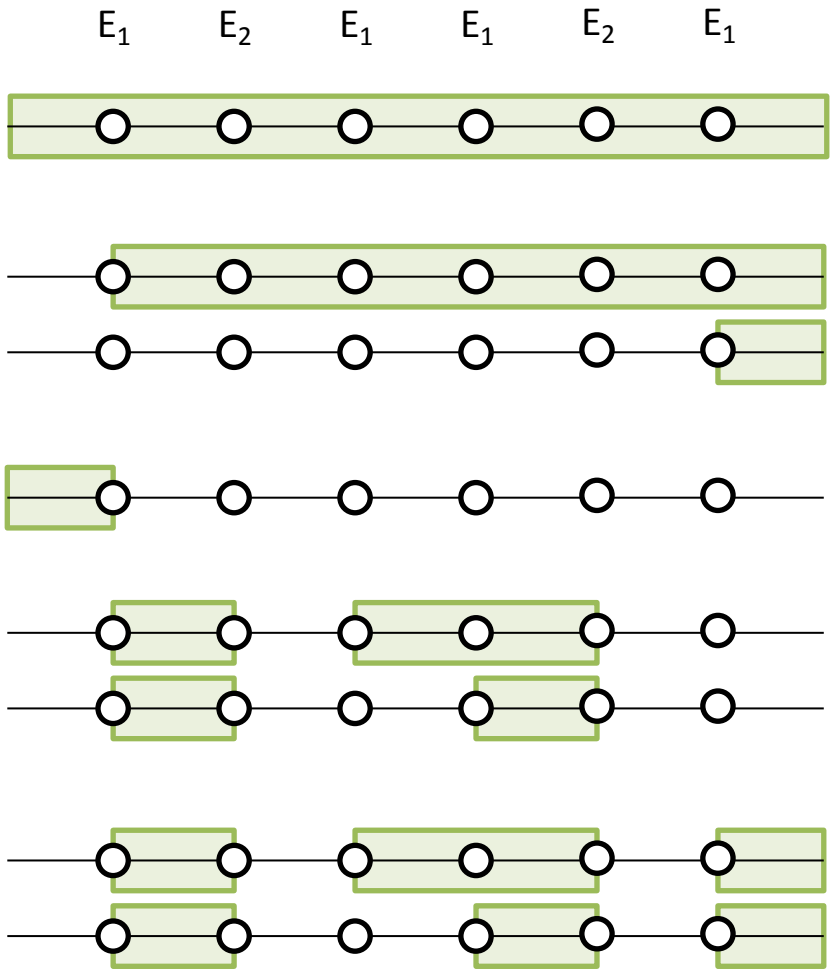
- Property = Pattern + Scope
  - Pattern: describes occurrences or orderings of events
  - Scope: describes the observation window on which the pattern is supposed to hold

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

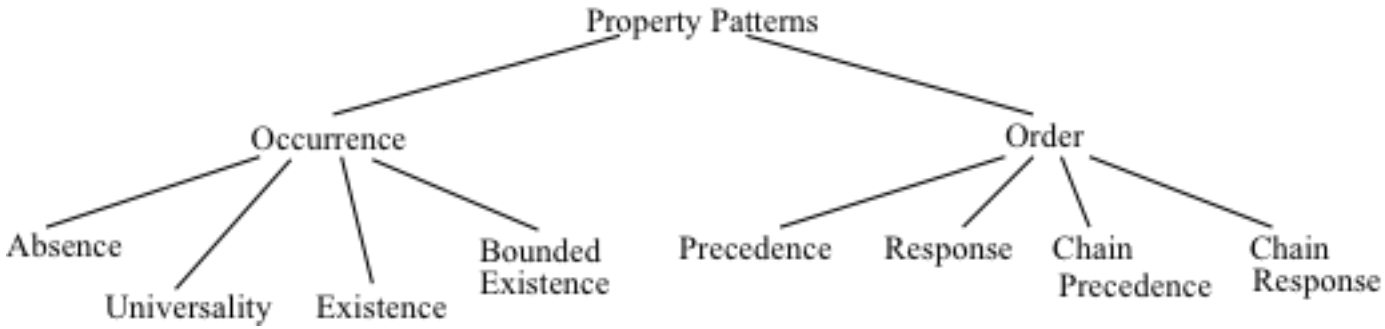
Temporal Properties in TOCL

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

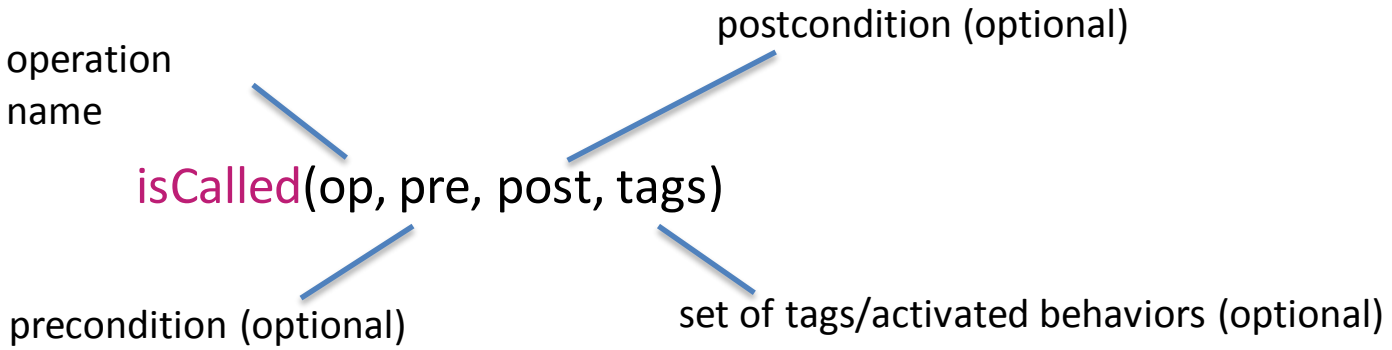


Patterns

- 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



becomesTrue(state predicate)

Evaluated to false before the event, and true after the event

Temporal Properties in TOCL

« Tickets can only be bought when the user is connected to the system. »

- Property 1
  never isCalled(buyTicket,{@AIM:BUY_Success})
  before isCalled(login,{@AIM:LOG_Success})

- Property 2
  never isCalled(buyTicket,{@AIM:BUY_Success})
  after isCalled(logout,{@AIM:LOG_Logout})
  until isCalled(login,{@AIM:LOG_Success})

- Property 3
  eventually isCalled(buyTicket,{@AIM:BUY_Success}) at least 0 times
  between isCalled(login,{@AIM:LOG_Success})
  and isCalled(logout,{@AIM:LOG_Logout})

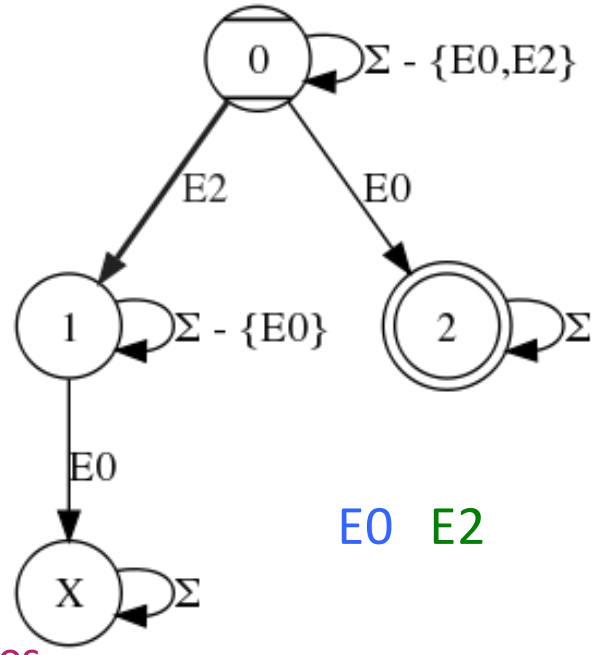
Temporal Properties in TOCL

« Tickets can only be bought when the user is connected to the system. »

- Property 1
  never isCalled(buyTicket,{@AIM:BUY_Success})
  before isCalled(login,{@AIM:LOG_Success})

- Property 2
  never isCalled(buyTicket,{@AIM:BUY_Success})
  after isCalled(logout,{@AIM:LOG_Logout})
  until isCalled(login,{@AIM:LOG_Success})

- Property 3
  eventually isCalled(buyTicket,{@AIM:BUY_Success}) at least 0 times
  between isCalled(login,{@AIM:LOG_Success})
  and isCalled(logout,{@AIM:LOG_Logout})



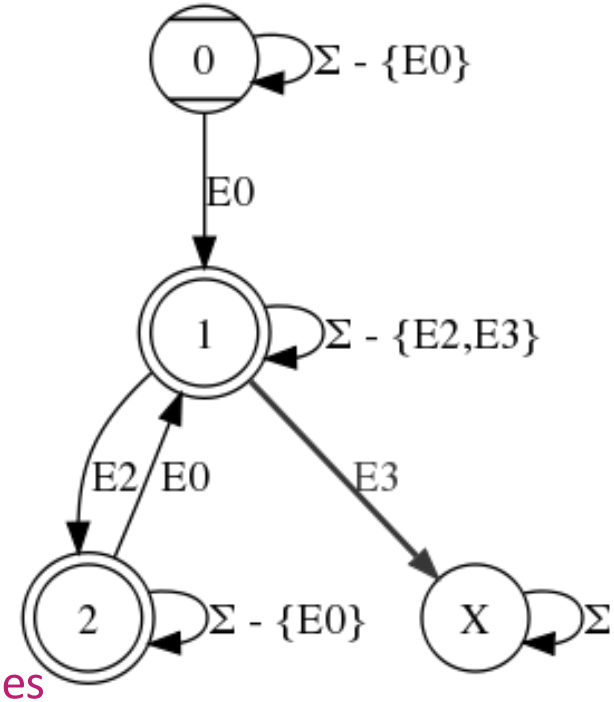
Temporal Properties in TOCL

« Tickets can only be bought when the user is connected to the system. »

- Property 1
  never isCalled(buyTicket,{@AIM:BUY_Success})
  before isCalled(login,{@AIM:LOG_Success})

- Property 2
  never isCalled(buyTicket,{@AIM:BUY_Success})
  after isCalled(logout,{@AIM:LOG_Logout})
  until isCalled(login,{@AIM:LOG_Success})

- Property 3
  eventually isCalled(buyTicket,{@AIM:BUY_Success}) at least 0 times
  between isCalled(login,{@AIM:LOG_Success})
  and isCalled(logout,{@AIM:LOG_Logout})



E0 E2 E3

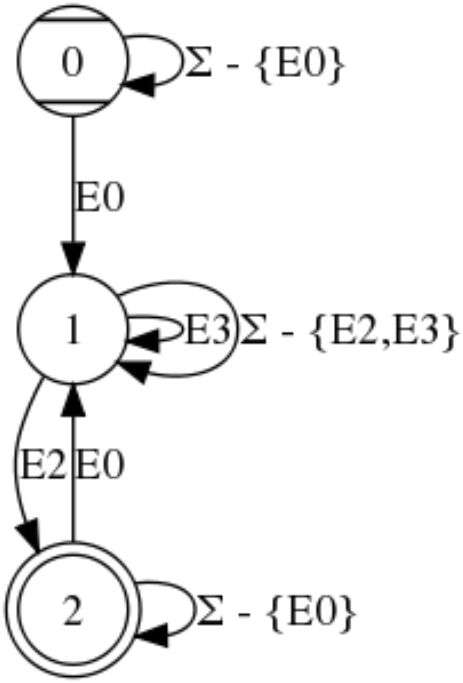
Temporal Properties in TOCL

« Tickets can only be bought when the user is connected to the system. »

- Property 1
  never isCalled(buyTicket,{@AIM:BUY_Success})
  before isCalled(login,{@AIM:LOG_Success})

- Property 2
  never isCalled(buyTicket,{@AIM:BUY_Success})
  after isCalled(logout,{@AIM:LOG_Logout})
  until isCalled(login,{@AIM:LOG_Success})

- Property 3
  eventually isCalled(buyTicket,{@AIM:BUY_Success}) at least 0 times
  between isCalled(login,{@AIM:LOG_Success})
  and isCalled(logout,{@AIM:LOG_Logout})



E0 E2 E3

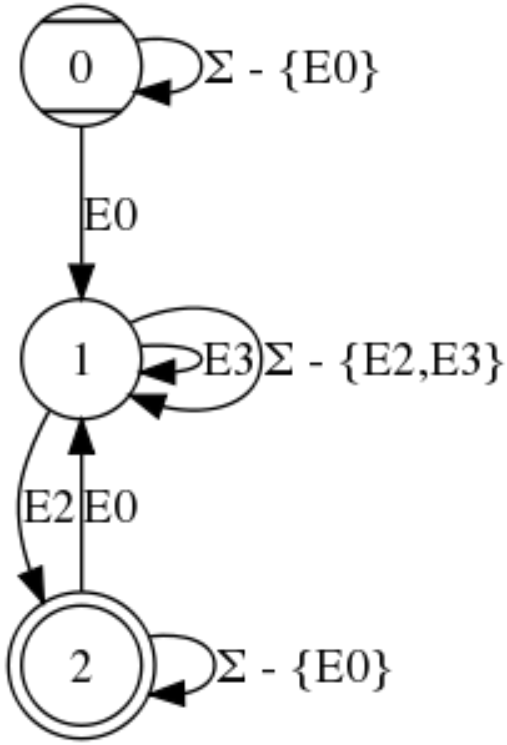
Outline

- Context and motivations
- Property pattern language
- Coverage criteria: nominal and robustness
- Experimental results
- Conclusion and perspectives

Using the properties for testing

- Existing automata coverage criteria are not appropriate

➔ all transitions are considered equally!



E0 = isCalled(login,{@AIM:LOG_Success})

E2 = isCalled(logout,{@AIM:LOG_Logout})

E3 = isCalled(buyTicket,{@AIM:BUY_Success})

Need to distinguish two different kinds of transition

⇒ α-transitions, labelled by events expressed in the property

⇒ Σ-transitions, the others

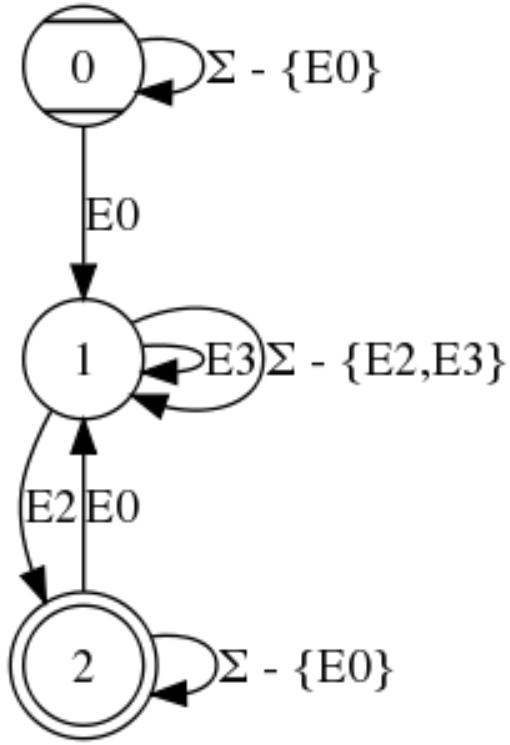
Also, the origin of all the transitions (scope/pattern) is known.

Using the properties for testing

- New coverage criteria for the property automata
  - alpha-transition coverage: coverage of the transitions labelled by events expressed in the property

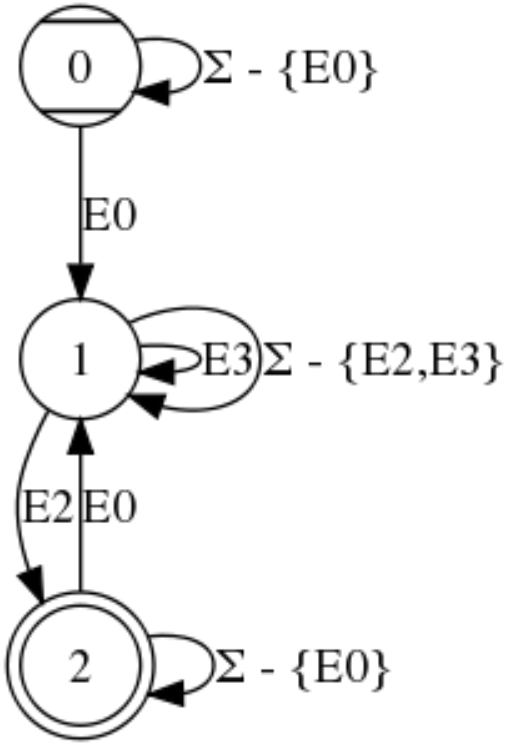
Transitions to cover:

(0, E0, 1)
(1, E3, 1)
(1, E2, 2)
(2, E0, 1)



Using the properties for testing

- New coverage criteria for the property automata
  - alpha-transition-pairs coverage: coverage of the pairs of transitions labelled by events expressed in the property



Pairs of transitions to cover:

< (0, E0, 1) ; (1, E3, 1) >
< (1, E3, 1) ; (1, E2, 2) >
< (1, E2, 2) ; (2, E0, 1) >
< (2, E0, 1) ; (1, E3, 1) >
< (2, E0, 1) ; (1, E2, 2) >

Important: strict successions of α-transitions are not required (intermediate Σ-transitions are allowed)

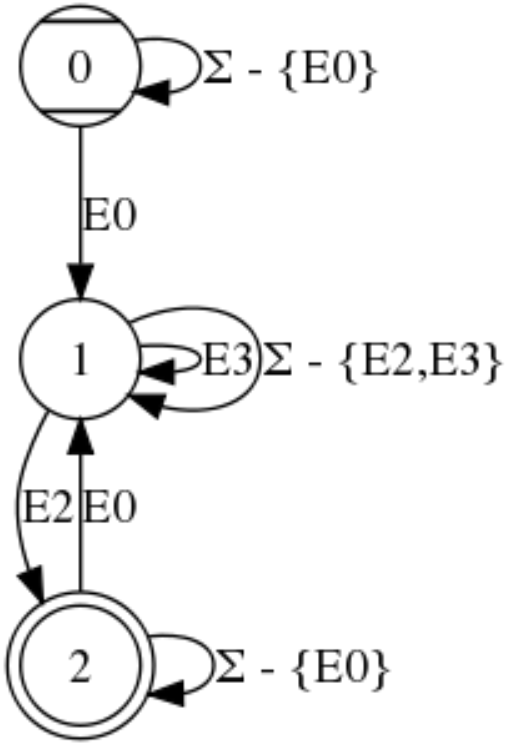
Using the properties for testing

- New coverage criteria for the property automata
  - k-pattern coverage: coverage of the iterations of the pattern

All pattern-loops have to iterated between 0 and k times.

Applicable to « repeatable » patterns:

- precedes
- follows
- eventually at least n times (if n >= k)



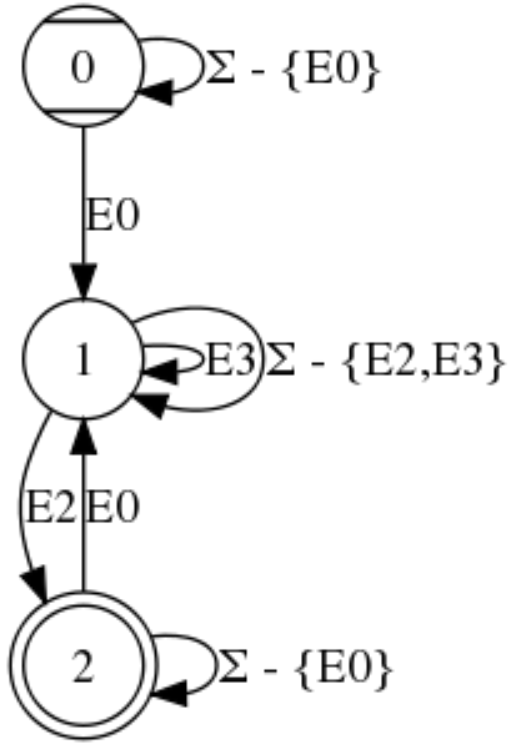
Using the properties for testing

- New coverage criteria for the property automata
  - k-scope coverage: coverage of the iterations of the scope

All scope-loops have to iterated between 1 and k times.

Applicable to « repeatable » scopes:

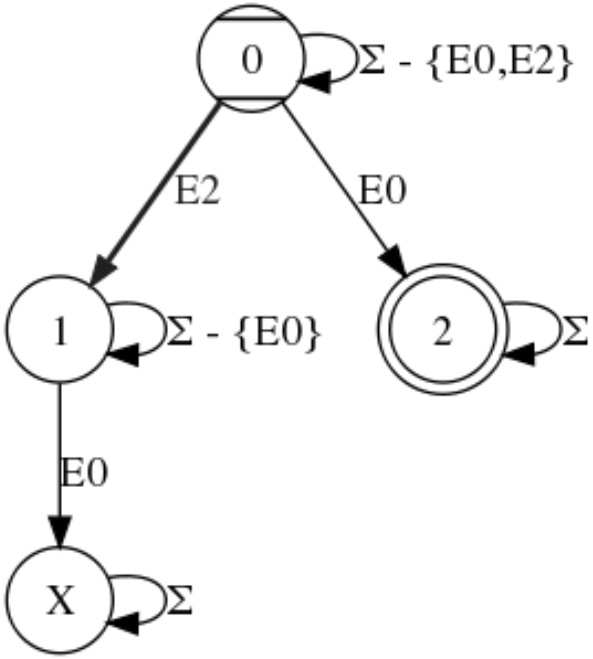
- between
- after… until…



Notice: interesting paths end on a final state of the automaton

Using the properties for testing (cont'd)

Case of transitions leading to the error state



Can not be activated if we assume that the model satisfies the property (which is supposed to be the case)

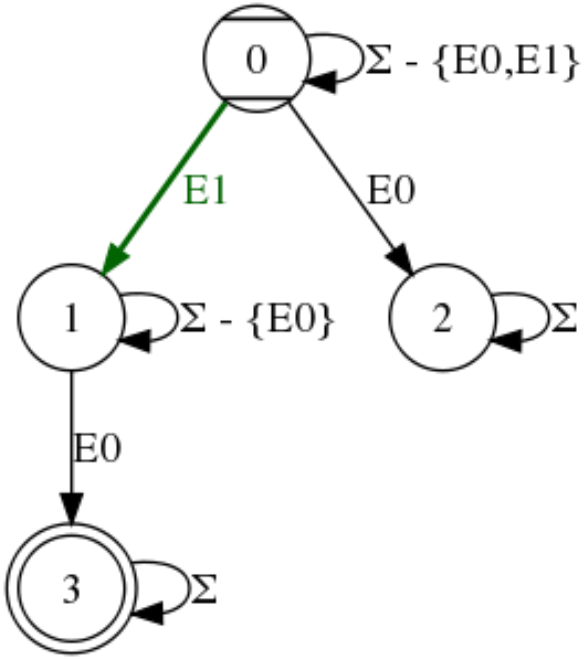
New coverage criteria are inefficient…

➔ Specific criterion to test the robustness of the system w.r.t. the property

never isCalled(buyTicket,{@AIM:BUY_Success})
before isCalled(login,{@AIM:LOG_Success})

Using the properties for testing (cont'd)

Coverage criterion: robustness



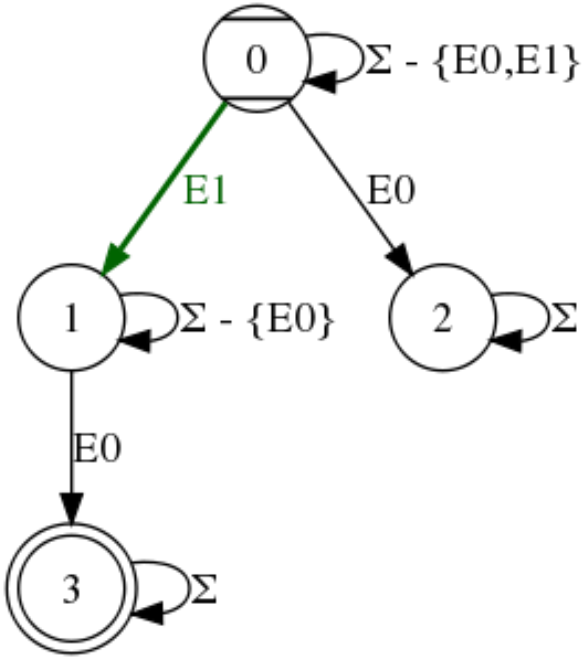
Modification of the automaton:
- the error state becomes the final state
- the event labelling the faulty transition is mutated/weakened to be made activable

Possible mutations:
- deletion/negation of predicates (pre/post)
- deletion/change of tags

E1 : isCalled(buyTicket,{@AIM:BUY_Success}) ➔ isCalled(buyTicket)

Using the properties for testing (cont'd)

- Two possible uses for these coverage criteria
  - Measure the quality of a test suite
  - Generate test scenarios



Functional test suite (computed using CertifyIt)
sut.buyTicket(TITLE2)

Test scenario:
(Σ - {E0,E1})* . E1 . (Σ - {E0})* . E0

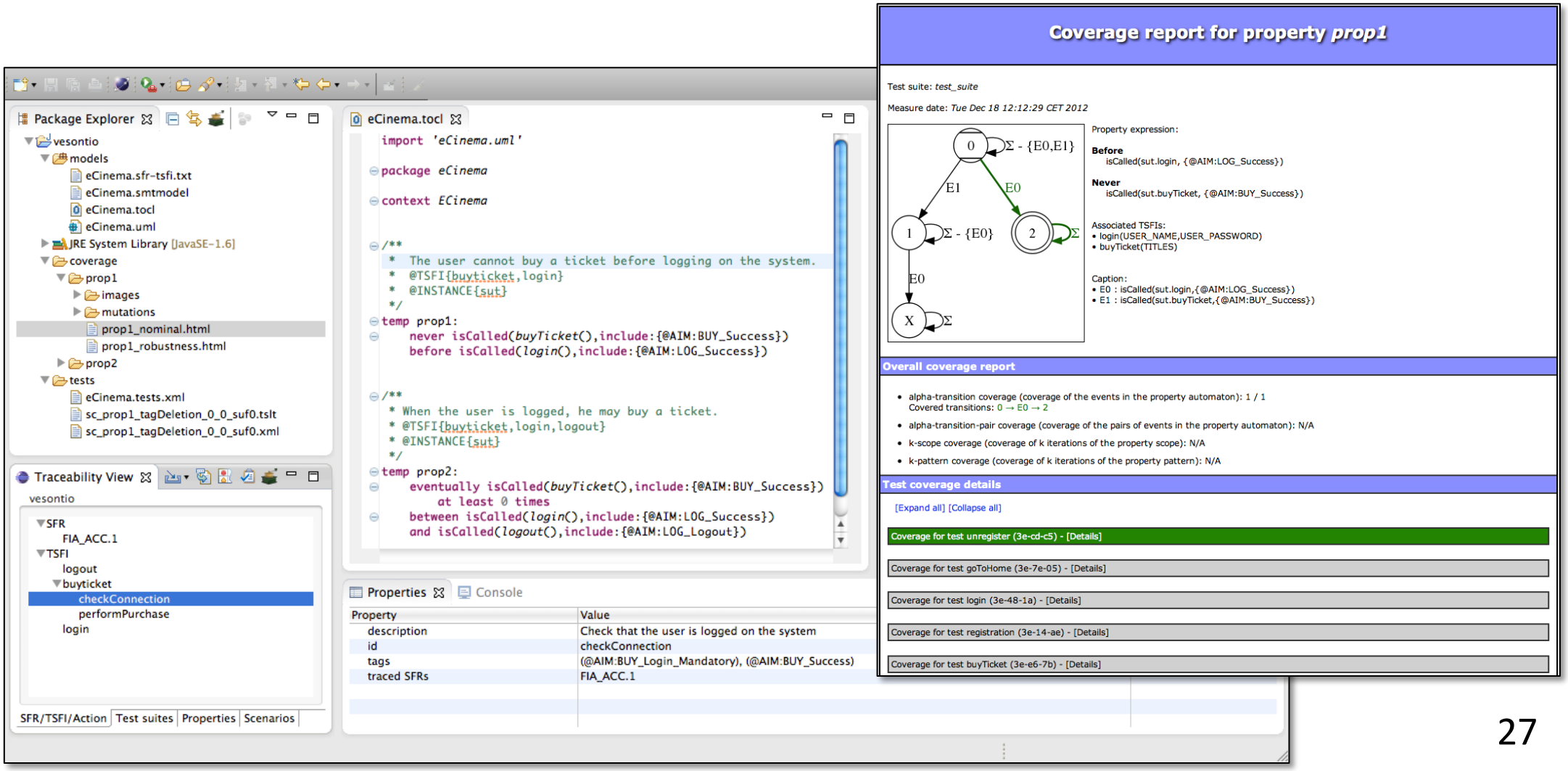
Corresponding test case:
sut.buyTicket(TITLE2);
sut.login(REGISTERED_USER, REGISTERED_PWD)

Outline

- Context and motivations
- Property pattern language
- Coverage criteria: nominal and robustness
- Experimental results
- Conclusion and perspectives

Experimental results

- Development of an Eclipse plug-in to support the approach



Experimental results

1st experiment: evaluation during industrial projects

- ANR TASCCC* – validation of smart cards security mechanisms for common criteria evaluation, in partnership with Smartesting, Gemalto (among others)
- ANR OSEP* – validation of cryptographic components, in partnership with Smartesting and the Armaments Procurement Agency

- Evaluation procedure
  - Start with an existing functional model and test suite (CertifyIt)
  - Design test properties for the considered models (3 case studies, 3-4 properties each)
  - Measure the property coverage criteria satisfaction

*funded by the French National research agency

Experimental results

Conclusions of the study

- Language is easy to learn and use to design test properties
  - however, sometimes validation engineers tend to write test cases instead of test properties ➔ unsatisfied 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

Experimental results

2nd experiment: evaluation of the error detection capabilities (robustness)

- Process:
  - Design 6 properties for the eCinema model
  - Complete the CertifyIt test suite to satisfy the robustness coverage criterion
  - Perform mutations on the model using the following mutation operators
    - SSOR : Simple Set Operator Replacement
    - SNO : Simple expression Negation Operator
    - SAF : Stuck-At-False
    - AD : Action Deletion
  - Evaluate how many mutants are killed by the tests, and compare to the initial TS

Experimental results

Test suites	Property-Based Testing				Smartesting CertifyIt			
Mutations / Verdicts	C-NE	NC-NE	NC-E	C-E	C-NE	NC-NE	NC-E	C-E
SSOR	2	1	1	2	4	1	1	
SNO		28	2			28		2
SAF		31	1			31	1	
AD	7	12		4	15	8		

Conformance (C)/Non-Conformance (NC): determined using basic observations (comparison of outputs)
Not in Error (NE), in Error state (E): determined by monitoring the states reached on the property automaton

C-NE Conform, not reaching the error state of the automaton (eq. mutant or mutant that could not be observed)

NC-NE Not-Conform, not reaching the error state (killed mutant, but not because it violated the property)

NC-E Not-Conform, and reaching the error state (killed mutant, violation of the property, detected by basic observations)

C-E Conform, but reaching an error state (unkilled mutant that violated the property, not detected by basic observations)

Experimental results

Test suites	Property-Based Testing				Smartesting CertifyIt			
Mutations / Verdicts	C-NE	NC-NE	NC-E	C-E	C-NE	NC-NE	NC-E	C-E
SSOR	2	1	1	2	4	1	1	
SNO		28	2			28		2
SAF		31	1			31	1	
AD	7	12		4	15	8		

Conformance (C)/Non-Conformance (NC): determined using basic observations (comparison of outputs)
Not in Error (NE), in Error state (E): determined by monitoring the states reached on the property automaton

Our approach is able to:

1. build test cases that make violations of a property observable
2. build test cases that consist in operations leading to a violation of the property
3. build new test cases that improve the error detection capabilities

Outline

- Context and motivations
- Property pattern language
- Coverage criteria: nominal and robustness
- Experimental results
- Conclusion and perspectives

Conclusion

- We have proposed in this paper:
  - a property-based testing approach using property patterns
  - associated coverage criteria (nominal or robustness)

- Useful for:
  - evaluating a test suite w.r.t. the property
  - test selection, to complete a functional test suite

Future works

- Improvement of the test generation process
  - Combinatorial explosion of test targets
  - Unfolding of test scenarios

- Integrate it as a plug-in for Smartesting CertifyIt

- Experiment at a larger scale
  - national project with Armaments Procurement Agency

Thanks for your attention!

Questions?

Video demo? flash me!

Projects websites:

http://disc.univ-fcomte.fr/TASCCC

http://osep.univ-fcomte.fr