Automatic Generation of Test Models and Properties from UML Models with OCL Constraints

Miguel A. Francisco[1] Laura M. Castro[2]

[1]Interoud Innovation S.L. (Spain) – miguel.francisco@interoud.com

[2]MADS Group – Dept. Computer Science – University of A Coruña (Spain) – lcastro@udc.es

Workshop on OCL and Textual Modelling, 2012, Innsbruck

Introduction

- Testing is a key activity in software development.
- Unfortunately, testing activities are often skipped.
- Some reasons why tests are not implemented are:
  - Testing is a very tedious task: developers say that they do not have time to test their systems.
  - Lack of good and complete testing tools and methodologies: developers have to use a different approach for each specific system to test.
- Our goal: try to mitigate these two issues so that testing activities are carried out.

Objectives

- Present a black-box testing approach that mitigates the described problems:
  - Testing is very tedious task.
  - Lack of good and complete testing tools and methodologies.
- What features should our approach have to mitigate these problems?
  - Testing is very tedious task ⇒ **Write less, test more**.
  - Lack of good and complete testing tools and methodologies ⇒ **Implementation-independent approach**.
- Searching for existing approaches with those features...

Write less, test more ⇒ Property-Based Testing

- In a Property-Based Testing approach, the test suite is a set of properties that the system under test must hold.
- Properties are declarative statements that describe the system under test requirements.
- Property-Based Testing tools implement algorithms to generate test cases from properties.

Example

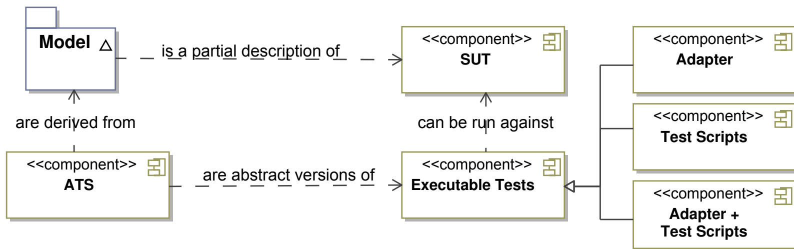
```
remove(I: Integer, L: List<Integer>):List<Integer>
```

The `remove` operation deletes all occurrences of the integer `I` from a list of integers `L`:

$$\forall \{I, L\} \in \{int(), list(int())\},\ I \notin remove(I, L)$$

Implementation-independent approach ⇒ Model-Based Testing

- Model-Based Testing approaches use a model to describe the system under test.
- The model is a schematic description of a system, independent from the system implementation.
- Model-Based Testing tools use the model as an input to generate an abstract test suite automatically.
- The abstract test suite must be transformed into an executable test suite.

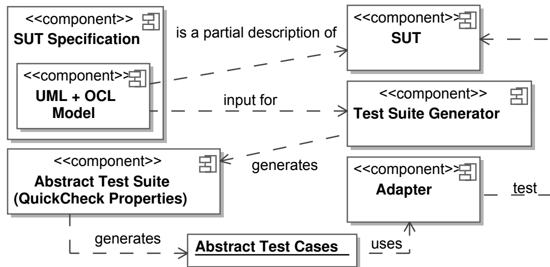


Combining Model-Based Testing with Property-Based Testing

- Our approach consists in combining Property-Based Testing with Model-Based Testing to get a new implementation-independent approach based on properties:
  - The system under test is described using a **UML** model with **OCL** constraints.
  - The generated abstract test suite is composed by a set of properties written for **QuickCheck**.

Architecture

- A *Test Suite Generator*, that uses **Dresden OCL** to parse the **UML+OCL** model, analyses the model and generates an abstract test suite, composed by a set of properties written for **QuickCheck**.
- This approach takes advantage of QuickCheck capabilities to generate test cases from those properties.



- We use an adapter to concretize the generated abstract test suite, transforming the abstract test suite into real invocations to the system under test.

A Lists Library

UML Model

Lists

+add(e : Integer, l : Integer [0..*]) : Integer [0..*]
+delete(e : Integer, i : Integer [0..*]) : Integer [0..*]
+sum(l : Integer [0..*]) : Integer
+prod(l : Integer [0..*]) : Integer
+min(l : Integer [0..*]) : Integer
+max(l : Integer [0..*]) : Integer

OCL Constraints

```
...
context ListUtils::max(l: Sequence(Integer)): Integer
pre not_empty: l->notEmpty()
post max: l->forAll(x : Integer | result >= x) and
          l->includes(result)
...
```

A Lists Library

OCL Constraint

```
context ListUtils::max(l: Sequence(Integer)): Integer
pre not_empty: l->notEmpty()
post max: l->forAll(x : Integer | result >= x) and
          l->includes(result)
```

A Lists Library

OCL Constraint

```
context ListUtils::max(l: Sequence(Integer)): Integer
pre not_empty: l->notEmpty()
post max: l->forAll(x : Integer | result >= x) and
          l->includes(result)
```

QuickCheck Property

```
prop_max_max()->
    ?FORALL(L,
        ?SUCHTHAT(L, ocl_gen:gen_sequence_integer(),
            begin
                ocl_seq:not_empty(L)
            end
        ),
        begin
            Result = listUtils:max(L),
            (ocl_seq:forAll(fun(X) -> (Result >= X) end, L)
                andalso ocl_seq:includes(Result, L))
        end).
```

A Lists Library: Preconditions

OCL Constraint

```
context ListUtils::max(l: Sequence(Integer)): Integer
pre not_empty: l->notEmpty()
post max: l->forAll(x : Integer | result >= x) and
          l->includes(result)
```

QuickCheck Property

```
prop_max_max()->
    ?FORALL(L,
        ?SUCHTHAT(L, ocl_gen:gen_sequence_integer(),
            begin
                ocl_seq:not_empty(L)
            end
        ),
        begin
            Result = listUtils:max(L),
            (ocl_seq:forAll(fun(X) -> (Result >= X) end, L)
                andalso ocl_seq:includes(Result, L))
        end).
```

A Lists Library: Postconditions

OCL Constraint

```
context ListUtils::max(l: Sequence(Integer)): Integer
pre not_empty: l->notEmpty()
post max: l->forAll(x : Integer | result >= x) and
          l->includes(result)
```

QuickCheck Property

```
prop_max_max()->
    ?FORALL(L,
        ?SUCHTHAT(L, ocl_gen:gen_sequence_integer(),
            begin
                ocl_seq:not_empty(L)
            end
        ),
        begin
            Result = listUtils:max(L),
            (ocl_seq:forAll(fun(X) -> (Result >= X) end, L)
                andalso ocl_seq:includes(Result, L))
        end).
```

A Lists Library: Test Suite Execution

Test Case	Precondition	Max	Postcondition
[]	false	–	–
[0]	true	0	true
[0, 1]	true	1	true
[2]	true	2	true
[1]	true	1	true
[-4,-2]	true	0	false

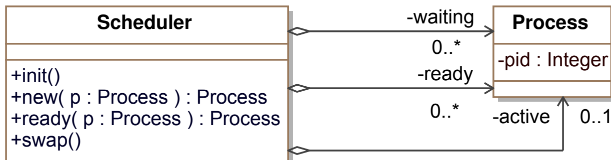
- The test case `[-4,-2]` does not satisfy the postcondition because the result `0` is not a member of the list `[-4,-2]`.
- QuickCheck shrinks to the smallest counterexample that causes the same error: `Shrinking..(2 times) [-1]`
- The implementation of the `max` operation fails when all the elements of the list are negative integers.

The OCL @pre operator

- OCL allows one to use more complex operators: `@pre`
- The `@pre` operator references a value in a postcondition before the operation execution.
- The test suite must store the value of `@pre` references before executing the operations to test.
- QuickCheck supports to define state machines which is perfectly suited to this situation.
- If the OCL specification uses the `@pre` operation, the *Test Suite Generator* will generate a QuickCheck State Machine.

A Scheduler

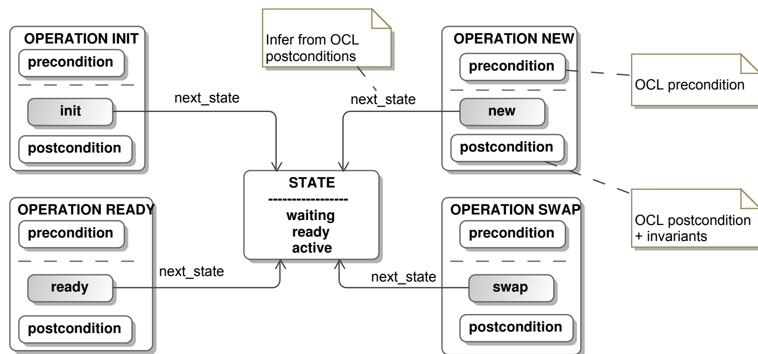
UML Model



OCL Constraints (with @pre operator)

```
context Scheduler
...
context Scheduler::New(p:Process)
pre: p <> active and
       not (ready->union(waiting))->includes(p)
post: waiting = (waiting@pre->including(p)) and
        ready = ready@pre and active = active@pre and
        result = p
...
```

A Scheduler: QuickCheck State Machine



- Identify the operations to test: `init`, `new`, `ready`, `swap`.
- Define the data store in the state: `active`, `ready`, `waiting`.
- Define the `precondition`, `postcondition` and `next_state` for each operation to test.

A Scheduler: Preconditions and Postconditions

OCL Specification

```
context Scheduler::New(p:Process)
pre: p <> active and not (ready->union(waiting))->includes(p)
post: waiting = (waiting@pre->including(p)) and
      ready = ready@pre and active = active@pre and result = p
```

QuickCheck Precondition

```
precondition(S, {call, scheduler, new, P})->
  (ocl:neq(P, S#ts.active) andalso not(ocl_set:includes(P,
      ocl_set:union(S#ts.waiting, S#ts.ready))));
```

QuickCheck Postcondition

```
postcondition(Pre, After, {call, scheduler, new, P}, R)->
  ocl_set:eq(After#ts.waiting,
      ocl_set:including(P, Pre#ts.waiting)) andalso
   ocl_set:eq(After#ts.ready, Pre#ts.ready) andalso
   ocl:eq(After#ts.active, Pre#ts.active) andalso
   ocl:eq(R, P) andalso invariant(Pre, After);
```

A Scheduler: Next State

- QuickCheck State Machines require the definition of the `next_state` functions to indicate how each operation modifies the state.

OCL Specification

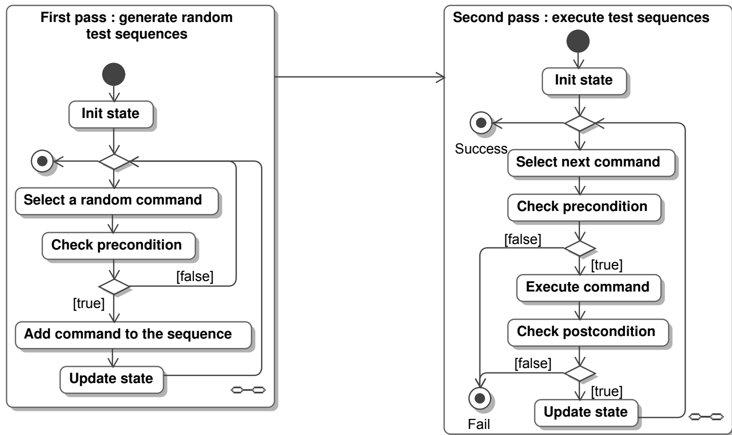
```
context Scheduler::New(p:Process)
pre: p <> active and not (ready->union(waiting))->includes(p)
post: waiting = (waiting@pre->including(p)) and
      ready = ready@pre and active = active@pre and
      result = p
```

QuickCheck Next State

```
next_state(S, R, {call, scheduler, new, P})->
    S#ts {
        active = S#ts.active,
        ready = S#ts.ready,
        waiting = ocl_set:including(P, S#ts.waiting)
    };
```

A Scheduler: Test Suite Execution in Two Passes

- QuickCheck generates random sequences of operations in a first pass checking the preconditions and modifying the state using the `next_state` functions.
- QuickCheck executes the generated random sequences of operations in a second pass checking the preconditions and the postconditions ⇒ **offline testing**.



Dynamic Preconditions

- Sometimes the `next_state` function requires the result of the operation to test to fill the state.
- For example, which will be the next `active` process after executing the `swap` operation?

```
context Scheduler::swap()
pre: active <> null
post: if ready@pre->isEmpty() then
         (active = null and ready = Set{})
      else (ready@pre->includes(active) and
            ready = ready@pre->excluding(active))
      endif and
      waiting = waiting@pre->including(active@pre)
```

- Therefore, the `active` state variable can not be used in the first pass tests sequences generation.
- In this case, preconditions will be only checked dynamically in the second pass ⇒ **online testing**.

Online Testing vs. Offline Testing

- Online testing with dynamic preconditions can be used in all cases.
- However, QuickCheck's shrinking capabilities only works with offline testing, i.e., with the two-pass algorithm.
- QuickCheck's shrinking capabilities provide smaller counterexamples that, causing the same error as the original failing test case, make error debugging and fixing easier.
- Therefore, offline testing should be used if possible, even though online testing could be always used.
- This is taken into account by the *Test Suite Generator*, and it uses offline testing if possible.

Conclusions

- Our approach transforms a UML model with OCL constraints into properties to be used as an implementation-independent test suite.
- Thus, it combines Model-Based Testing with Property-Based Testing: it is a Model-Based Testing approach that takes advantage of a Property-Based Testing tool to generate test cases from generated properties.
- The approach was used in several situations:
    - Stateless components: QuickCheck properties.
    - Fully specified stateful components: QuickCheck State Machine with offline test cases generation.
    - Partially specified stateful components: QuickCheck State Machine with online test cases generation.

Future Work

- Improve the support for more OCL features: OCL messages and more built-in operations.
- Improve data generators: not generate only random data of the given data type, but generate data to combine positive and negative testing (for example, processes that are in the list of waiting processes, processes already generated, new processes, etc).
- Extend this approach to other kind of testing, specifically, integration testing.

Automatic Generation of Test Models and Properties from UML Models with OCL Constraints

Miguel A. Francisco[1] Laura M. Castro[2]

[1]Interoud Innovation S.L. (Spain) – miguel.francisco@interoud.com

[2]MADS Group – Dept. Computer Science – University of A Coruña (Spain) – lcastro@udc.es

Workshop on OCL and Textual Modelling, 2012, Innsbruck