A subset of precise UML for Model-based Testing

F. Bouquet

University of Besançon
LIFC, 16 route de Gray 25030
Besançon, France

bouquet@lifc.univ-fcomte.fr

C. Grandpierre, B. Legeard, F. Peureux, N. Vacelet

LEIRIOS
18 rue Alain Savary
25000 Besançon, France

{grandpierre, legeard, peureux, vacelet}@leirios.com

M. Utting

Department of Computer Science
Private Bag 3105 Hamilton – New-Zealand

marku@cs.waikato.ac.nz

ABSTRACT

This paper presents an original model-based testing approach that takes a UML behavioural view of the system under test and automatically generates test cases and executable test scripts according to model coverage criteria. This approach is embedded in the LEIRIOS Test Designer tool and is currently deployed in domains such as Enterprise IT and electronic transaction applications. This model-based testing approach makes it possible to automatically produce the traceability matrix from requirements to test cases as part of the test generation process. This paper defines the subset of UML used for model-based testing and illustrates it using a small example.

Keywords

Model-Based Testing, UML, OCL

1. INTRODUCTION

There are several views of model-based testing: for example the generation of test cases from an environment model (see e.g. [1] based on a usage profile statistical model) or the generation of test cases with oracles from a behaviour model (see [2] for more details on model-based testing approaches). In this paper, we define model-based testing as *a process to automatically generate test cases and executable test scripts from a behaviour model of the system under test* (SUT). The model formalizes the expected behaviour to be tested on the SUT. Generated tests are sequences of operation invocations on the SUT, and include the expected output (the oracle information), so that test verdicts can be assigned automatically during test execution [3]. Moreover, each behaviour of the SUT model can be labelled with a particular requirement identifier, which makes it possible to generate a traceability matrix that links the generated tests to the initial requirements of the informal specification [4].

The Unified Modeling Language[1](UML) is widely used as a modelling support for model-based testing. There are several reasons for this interest. Firstly, UML provides a large set of diagrammatic notations for modelling purposes, with several complementary representations. A static representation (i.e. class diagrams) is used to model the points of control and observation of the SUT and the data that represents the abstract state of the SUT. A dynamic representation (e.g. state diagrams or activity diagrams) is used to model the expected behaviour of the SUT. Secondly, the Object Constraint Language (OCL [5]) associated with UML makes it possible to have precise models

[1] UML is a notation from the Object Management Group – See www.omg.org

– this means that the expected behaviour can be formalized using OCL. Thirdly, UML is the de-facto industrial standard for modelling enterprise IT applications; most software engineers have had some first level training on UML – this is an important point to facilitate the acceptance of a disruptive process such as model-based testing.

However, UML contains a large set of diagrams and notations, defined in a flexible and open-ended way using a meta-model[2] and with some freedom allowed for different interpretations of the semantics of the diagrams by different UML tools. So for practical model-based testing it is necessary to select a subset of UML and clarify the semantics of the chosen subset so that model-based testing tools can interpret the UML models.

The paper defines a subset of UML 2.1 (the latest version of UML) for model-based testing purposes. This subset allows *formal* behaviour models of the SUT to be designed, which can be mechanically interpreted to generate test suites. The subset uses class, instance and state diagrams, plus OCL expressions. Such UML models are used as input for a model-based test generator, called LEIRIOS Test Designer, that automates – using theorem prover technology – the generation of test sequences, covering each behaviour in the model.

This paper is organized as follows. Section 2 introduces the subset of UML and OCL we are proposing. Section 3 shows on a small Stack example how this notation can be used to develop a behaviour model for test generation purposes, and exhibits the generated test cases. Section 4 discusses some related work and Section 5 gives conclusions. One of the main contributions of the paper is to identify a subset of UML that is expressive enough to model a variety of industrial applications, and has a clear semantics that allows executable models to be written for test generation purposes. Another contribution is a novel active/passive interpretation of OCL expressions that allows OCL to be used as an action language in UML state machines and class diagrams.

2. A UML SUBSET

In this section, we define the UML subset we propose for model-based testing. The goal of this subset is to offer precise, necessary and sufficient modelling features to design behaviour models for test generation purposes. We call this subset UML-MBT.

2.1 UML 2.1 diagrams

The proposed subset for model-based testing is based on three diagrams: UML *class diagrams* (to model the points of control and observation of the SUT), *object diagrams* (to define test data), *state-machines and OCL*[3] (to model dynamic behaviour of the SUT). UML offers several other diagrams, but these are sufficient to design comprehensive, precise and interpretable models to test finite state systems with our tool. To model dynamical behaviour, we use state machine rather than UML sequence diagrams because state machines allow richer behaviours to be specified, with better support for loops and alternative paths.

[2] A meta-model is a language to describe the domains of applicative models and to define their semantics
[3] See UML 2.0 OCL Specification. http://www.omg.org/docs/ptc/

This section defines the subsets of these UML diagrams that are used to design UML-MBT models for automated test generation.

2.1.1 Class diagrams

The UML class diagram is the static view of the model. It describes the abstract objects of the system and their dependencies. The UML elements available to model the class diagram are the following elements:

• *Classes* define the types of the objects of the system. Inheritance is not yet implemented.

• Reflexive and binary *associations,* represent dependencies between classes. The available multiplicities are *0..1, m, n..* and n..m* with *m* and *n* integers such as *0≤n<m*. Association classes are not implemented.

• *Enumeration classes,* composed only of literals, are used to model static types.

• Class *attributes* define the state variables of the system. The supported types are integer, Boolean and enumeration types. For integers, a restricted domain is recommended for testing purposes (for example the integer interval [-32668, 36767]). Object types are represented using 1-1 associations, rather than attributes.

• *Operations* model the actions owned by an object. An operation can be defined with parameters (input and output) that can be typed as Boolean, integer, enumeration literal or object. OCL preconditions and postconditions can be used to formalise the behaviour of an operation.

2.1.2 Object diagrams

The UML object diagram lists the concrete objects used to compute test cases, and defines the initial state of the model. Each object diagram must be an instantiation of the associated class diagram. Notice the following restriction: objects can not be created or deleted dynamically by the actions designed in the model. So all objects used to describe the life cycle of the system must be defined in the object diagram. The dynamic creation (resp. deletion) of entities in the concrete system is simulated by creation (resp. deletion) of links between objects in the UML model.

The following UML elements can be used in object diagrams:

• *Objects,* or class instances, are the concrete objects of the system that are used in the generated tests. Every *slot* – or attribute instance – of every object must have a value.

• *Links,* or association instances, define the dependencies between the objects in the initial state of the system.

2.1.3 State-machines

State-machine diagrams are an optional part of a UML-MBT model. They are used to model the dynamic behaviour of the SUT as a finite state transition system.

The UML state machines used in UML-MBT may contain the following elements:

• initial (and optionally final) states,

• simple states, used to define the different system states of the SUT lifecycle,

• single transitions, used to model SUT actions. Transitions may be between two states, reflexive, or internal. A transition is composed of :

- an optional event (optionally defined with input parameters that can be typed as Boolean, integer, enumeration literal or object) that triggers the transition,
- an optional guard that is a Boolean expression used to determine whether or not the transition can fire,
- an optional action that updates the model data.

The semantics of state machine processing is based on the UML run-to-completion processing assumption. Run-to-completion processing means that an event occurrence can only be taken from the pool of operations declared in the class diagram. Moreover, this event can be dispatched only if the processing of the previous current occurrence is fully completed (to avoid concurrency conflicts during the processing of events). The processing of a single event occurrence by a state machine is known as a run-to-completion step. Before commencing on a run-to-completion step, a state machine has to be in a stable state configuration (a state in which no more transitions can be fired without external events). Thus, an incoming event will never be processed while the state machine is in some intermediate and inconsistent situation. A run-to-completion step can also be viewed as a complex state transition between two stable states of the state machine.

Just as OCL is used in class diagrams, to formalise the expected behaviour of class operations, OCL is also used within state machines to formalize transitions between states. – the guards and the effects of transitions are expressed as OCL predicates.

2.2 OCL 2.0 subset

To be able to execute transition actions and operation postconditions, UML-MBT uses an operational interpretation of OCL expressions used in such contexts. For example the OCL expression *self.attribute=true* can be used in two different contexts: a *passive* and an *active* context. A passive context is used to express constraints on the system under test, while an active context is used to express state changes in the model. So the expression *self.attribute=true* is interpreted and evaluated as a standard Boolean expression in a passive context. In an active context it is interpreted as an assignment of the value *true* to the Boolean state variable *attribute*.

We found it necessary to introduce this active/passive operational interpretation of OCL into UML-MBT because of the lack of *frame* information in OCL. That is, an OCL postcondition such as attribute1=attribute2 states that the two attributes must be equal after the operation, but does not specify the operational details of which attribute (attribute1 or attribute2 or both) was updated in order to satisfy the postcondition.

In this section, we explain how UML-MBT classifies each OCL Boolean expression as being either passive or active, and describe the meaning of each supported OCL operator in each context. This non-ambiguous interpretation of OCL expressions makes it possible to use OCL as an executable action language for model-based testing UML models.

The two interpretations (passive and active) were described briefly in [2], but are described in detail in Section 2.2.1 and Section 2.2.2. It should be noted that some OCL operators are allowed in both contexts so appear in both sections.

2.2.1 Passive OCL contexts

In UML-MBT, an OCL *passive expression* is an OCL expression that is used in a passive context. This includes operation preconditions, transition guards, decisions in conditional structures and all sub-expressions of other passive expressions. Passive expressions are used to *test* the state variables of a model -- they do not modify the model state. This section defines the set of all supported OCL passive expressions in UML-MBT.

2.2.1.1 Boolean operators

Table 1 lists the Boolean operators available in the UML-MBT set. In this table *p1* and *p2* are passive Boolean expressions.

Table 1. OCL Boolean operators

OCL notation	Operator	Result type
p1 = p2	equals	Boolean
p1 <> *p2*	not equals	Boolean
p1 or *p2*	disjunction	Boolean
p1 xor *p2*	excl. disjunction	Boolean
p1 and *p2*	*p1* and *p2*	Boolean
not *p1*	negation	Boolean

2.2.1.2 Integer operators

Table 2 lists the integer operators available in UML-MBT. In this table *i1* and *i2* are integer expressions.

Table 2. OCL integer operators

OCL notation	Operator	Operator
i1 = i2	equals	Boolean
i1 <> *i2*	not equals	Boolean
i1 < *i2*	lesser	Boolean
i1 > *i2*	greater	Boolean
i1 < *i2*	lesser	Boolean
i1 > *i2*	greater	Boolean
i1 <= *i2*	lesser or equal	Boolean
i1 >= *i2*	greater or equal	Boolean

i1 + *i2*	plus	Integer
i1 **–** *i2*	minus	Integer
- *i1*	unary minus	Integer
i1 * *i2*	multiplication	Integer
i1.**div(***i2***)**	division	Integer
i1.**abs()**	absolute value	Integer
i1.**mod(***i2***)**	modulo	Integer
i1.**max(***i2***)**	maximum	Integer
i1.**min(***i2***)**	minimum	Integer

2.2.1.3 Enumeration operators

Table 3 lists the enumeration operators available in UML-MBT. In this table *e1* and *e2* are enumeration literals.

Table 3. OCL enumeration operators

OCL notation	**Operator**	**Result type**
e1 **=** *e2*	equals	Boolean
e1 **<>** *e2*	not equals	Boolean

2.2.1.4 Object/Class operators

Table 4 lists the UML-MBT operators applicable to classes or objects (class instances). In this table *o1* and *o2* are objects and *c1* is a class.

Table 4. OCL class/objects operators

OCL notation	Operator	Result type
o1 = *o2*	equals	Boolean
o1 <> *o2*	not equals	Boolean
o1.oclIsUndefined()	is null	Boolean
c1.allInstances()	get all instances	Set

2.2.1.5 Collection operators

Table 5 lists the collection operators available in UML-MBT. In this table s1 and s2 are sets of objects and o1 is an object.

Table 5. OCL collection operators

OCL notation	Operator	Result type
s1 = *s2*	equals	Boolean
s1 <> *s2*	not equals	Boolean
s1->size()	size	Integer
s1->includes(*o1*)	includes	Boolean
s1->excludes(*o1*)	excludes	Boolean

s1->includesAll(*s2*)	includes all	Boolean
s1->excludesAll(*s2*)	excludes all	Boolean
s1->isEmtpy()	is empty set	Boolean
s1->notEmpty()	is not empty set	Boolean
s1->including(o1)	including	Set
s1->excluding(o2)	excluding	Set

2.2.1.6 Collection iterative operators

Table 6 lists the collection iterative operators available in UML-MBT. In this table *s1* is a set of objects, *p1* is a passive Boolean expression and *o1* must be the name of an association link in the class diagram. Note that the expression s1->collect(o1) can also be written more simply as s1.o1.

Table 6. OCL collection iterative operators

OCL notation	**Iterative operator**	**Result type**
s1->collect(*o1*)	collect	Set
s1->select(*p1*)	select	Set
s1->exists(*p1*)	exists	Boolean
s1->forAll(*p1*)	for all	Boolean
s1->any(*p1*)	any	Object

2.2.2 Active OCL contexts

In UML-MBT, an OCL *active expression* is an expression that is used in an active context. This includes operation postconditions, transition actions, action in the **then** or **else** part of a conditional structure, or a sub-expression of an active expression. Active expressions are used to change the values of state variables and to define values for the *return* parameter of operations. This section defines the set of all supported OCL active expressions in UML-MBT.

2.2.2.1 Assignment operator =

OCL uses the equality symbol to compare two elements (*e1=e2*). However, in an active context, we interpret this operator as an assignment operator. The left hand variable is assigned the value of the right hand expression. Thus this operator becomes non-commutative in an active context. For example the expression *self.attribute = true* sets the value of *self.attribute* to the Boolean value *true*. The assignment operator can be used to update any attribute value, any link and any set of links.

2.2.2.2 oclIsUndefined / isEmpty *operators*

In an *active* context these operators are used to delete links – association instances.

The operator *expr.oclIsUndefined(),* where *expr* refers to an association between classes with multiplicity 1 or 0..1, deletes any existing link and sets *expr* to null. Similarly, *expr.isEmpty(),* where *expr* refers to an association whose maximum multiplicity may be greater than one, deletes all the related links and sets *expr* to the empty set.

2.2.2.3 forAll *iterative operator*

The *active* expression *coll->forAll(expr)* operator applies the active expression *expr* to each object in the collection *coll*. This is similar to a loop in an imperative language.

2.2.2.4 and *operator*

In an active context the operator *and* acts as a separator between two *active* expressions.

2.2.2.5 if-then-else *structure*

This structure makes it possible to perform conditional execution of active OCL expressions.

The basic use is "***if*** *condition* ***then*** *action1* ***else*** *action2* ***endif"***, *w*here *condition* is a Boolean passive expression and *action1*, *action2* are active expressions.

All these active expression operators will be used in the model example given in section 3.

2.3 UML/OCL for MBT: key issues

The UML-MBT subset of UML defined in this paper needs specific interpretations in order to manipulate behavioural models and generate tests. We present here some key issues we address when generating tests with UML using the UML-MBT subset.

2.3.1 Model behaviours

The UML-MBT subset allows designing behavioural models. These behaviours are designed in the operation postconditions (in the class diagram) and in the transition actions (in state machines).

A set of consecutive actions – active expressions – defines the model behaviours. A conditional structure makes it possible to model alternative and complex behaviours in a single action or postcondition.

Example 1. Behaviours from operation postcondition

> Given the static operation *getType* – from a class *Triangle* – which returns the type of a triangle defined by its sides (*a*, *b* and *c*). This example is a version from the well-known example mentioned in [6]. The operation expressed with OCL notation is the following:
>
> **context:** Triangle::getType(a:Integer, b:Integer, c: Integer):TYPE
> **pre: a** >0 and b>0 and c>0
> **post:**

```
if  a+b<=c or a+c<=b or b+c<=a then
result = TYPE::NO_TRIANGLE
else
if a=b or b=c or a=c then
if a=b and b=c then
result = TYPE::EQUILATERAL
else
result = TYPE::ISOSCELES
endif
else
result = TYPE::SCALENE
endif
endif
```

In this postcondition of the "getType" operation, we clearly distinguish the four behaviours of the operation which define the four different triangle types.

The model behaviours allow generating tests on the basis of cause/effects defined via OCL expressions. We call *test target* a pair cause/effect that corresponds to a path in a post condition of an operation. More precisely, a test target is a pair defining an operation (or action linked to a transition) including the *effect* of the test target **and** one *target context* that makes it possible to produce the effect.

For the postcondition of the "getType" operation, the following test targets are computed:

Table 7. Test targets of the *getType* operation

Id	Target context	Target effect
1	**a+b<=c or a+c<=b or b+c<=a**	result= NO_TRIANGLE
2	not(a+b<=c or a+c<=b or b+c<=a) and (a=b or b=c or a=c) and **(a=b and b=c)**	result= EQUILATERAL
3	not(a+b<=c or a+c<=b or b+c<=a) and (a=b or b=c or a=c) and **not(a=b and b=c)**	result= ISOSCELES
4	not(a+b<=c or a+c<=b or b+c<=a) and **not(a=b or b=c or a=c)**	result= SCALENE

Some structural coverage criteria [7] can be applied to these targets contexts to create new derived test targets. For example, the Decision/Condition Coverage applied to the target 1 produces the 3 new test targets defined as follows (*target context* → *target effect*):

- a+b<=c → result=NO_TRIANGLE,

- a+c<=b → result=NO_TRIANGLE,

- b+c<=a → result=NO_TRIANGLE.

In addition, our interpretation of OCL makes it possible to increase or decrease the number of model behaviours, and so the number of test targets. The Boolean keywords *true* and *false* used in an *active* context allow tuning the test target generation. The *true* keyword used in an active context is interpreted to mean *skip* (that is, no change), while the *false* keyword is interpreted to mean infeasible behaviour, so no test targets will be produced for any path through an OCL active expression that contains false. These active interpretations of true and false are typically used in one branch of a conditional structure, to control test generation.

Table 8 shows the test targets generated from several examples of OCL conditional active expressions.

Table 8. Test targets from conditional structures

OCL expression	**Test targets**	
	Target context	*Target effect*
If cond **then** act1 **else** act2 **endif**	cond	act1
	not (cond)	act2
If cond **then** act1 **else** true **endif**	cond	act1
	not (cond)	skip
If cond **then** act1 **else** false **endif**	cond	act1

Notice that both branches of an OCL if-then-else structure must always be filled, which is why it is sometimes useful to use true or false in one branch.

2.3.2 OCL undefined value

Model-based testing is used to generate concrete tests from an abstract model. So an executable test must be defined with concrete values for each variable or parameter.

Now, OCL suggests the specific value *undefined* to qualify an expression without defined value. This undefined value is similar to the *null* value in Java. In OCL it can be tested with the special operator *oclIsUndefined()*.

An OCL expression is evaluated to the *undefined* value in the following cases:

• When the expression *coll->any(expr)* has no object to return, because there are no objects in *coll* that satisfy *expr*. That is, *coll->select(expr)* is empty.

• When the expression *exp.role* is applied to an empty association (that is, no link is defined between the object expressed by *exp* and the target object expressed by *role*).

• When a division or a modulo by zero occurs.

An OCL expression is *undefined* if it contains any subexpression whose value is *undefined*. That is, all operators are *strict* in their interpretation of undefined. Note that this is one difference from the usual OCL semantics for Boolean operators, which use a three-valued non-strict interpretation of undefined – the UML-MBT style is to use explicit if-then-else expressions in such cases.

This strict interpretation has an effect on the model behaviours and so on the test targets extracted from the model. Thus a test target for which the *target context* and/or the *target effect* is evaluated to *undefined* cannot be reached and so will give no test for the corresponding behaviour. The expression *if cond then action1 else action2 endif* generates the test targets t1 defined by *cond and action1* and t2 defined by *not(cond) and action2*. If *cond* is undefined both test targets are unreachable. If *action1* is undefined then t1 is unreachable. If *action2* is undefined then t2 is unreachable.

2.3.3 Specific ANY operator

The OCL *any* operator is used on a collection to obtain an arbitrary element of the collection. If several elements satisfy the *any* expression, the element is chosen non-deterministically. If no element respects the expression, then the *any* operator returns the undefined value. So our strict interpretation of *undefined* means that when no object satisfies an *any* operator, the corresponding behaviour will not be reachable.

In addition, the *any* operator is normally non-deterministic. The expression *coll->any(expr)* returns an arbitrary object of *coll* for which *expr* is true. However, to ensure reproducibility of test generation the execution of such an expression must always return the same object for the same test. We satisfy these requirements by taking the test context into account when choosing the object to be returned by an *any* expression. The *any* operator interpretation is illustrated in the following example.

Example 2. *any* operator interpretation

> Consider a class called *A* with an integer attribute named *attr*. Consider three instances, *a1, a2* and *a3,* of the class *A*, with *a1.attr=1, a2.attr=2* and *a3.attr=3.* Consider the following postcondition expressed in OCL.
>
> ```
> post:
>  let obj = A.allInstances()->any(attr>1) in
>         if obj.attr = 2 then
>                 result = MSG::MSG1
>         else
>                 if obj.attr > 2 then
>                         result = MSG::MSG2
>                 else
>                         result = MSG::MSG3
>                 endif
>         endif
> ```
>
> In this postcondition the first behaviour is reachable, because *any(attr>1)* is verified by instance a2, the second behaviour is also reachable, because *any(attr>1)* is verified by instance a3, but the last behaviour is unreachable. However if the *any* expression was *attr<1*, no behaviour would be reachable in this postcondition.
> The *any* operator can return different objects (a1 or a2 here), but always the same object for the same behaviour. In the example, *a2* is always given to reach the first behaviour; *a3* is always given to satisfy the second behaviour.

2.3.4 Requirements traceability

UML-MBT supports the expression of requirements that are external to the model (they usually come from the informal and often textual, specification of the system). A requirement can be related to any effect designed in the operation postconditions or in the transition actions. Such effects are also directly annotated in the OCL constraints with a specific identifier that refers to the expression of the related requirement.

Concretely a requirement is expressed with a specific form of comment block. The start and the end requirement markers are "/*@REQ:" and "@*/". Everything enclosed in this specific comment block is considered to be a declaration of requirements.

The *Stack* example of the next part illustrates the use of these requirement identifiers to achieve requirement traceability.

3. APPLICATION EXAMPLE

We propose in this section an example of the application of model-based testing with the UML-MBT set in order to generate tests from a specification modelled with UML/OCL.

In this example, the system under test is a chained stack of generic elements. The stack is loaded via a *push* operation and is emptied via a *pop* operation. The elements to push are randomly chosen from a pool of elements. The maximum size of the stack is the constant MAX. A list of functional requirements which must be assumed is given in Table 9.

In addition this example shows the different specific points presented in the previous section.

3.1 Requirements

We consider that the system under test has to satisfy the following requirements.

Table 9. Stack Requirements

Identifier	**Requirement description**
pool_empty	The pool can be emptied out by one operation
pool_fill	The pool can be completely filled by one operation
empty_stack_exception	A *pop* operation on an empty stack generates an exception
full_stack_exception	When the stack size equals MAX, a *push* operation on the stack generates an exception
random_element, automatic_delete	The stack is loaded with elements from the pool. The elements are loaded one by one and chosen randomly from the pool. This element is automatically deleted from the pool.
automatic_reinsertion	A popped element is automatically put into the pool.

3.2 The *Stack* Model

We present here a model of the Stack system, designed for test generation purposes. Some modelling choices are discussed too.

3.2.1 Class diagram

Figure 1 presents the class diagram. It depicts the different objects of the system under test and the dependencies between them.
We have three object types in the system. The stack is composed of chained elements taken from a pool. This pool contains a collection of elements.

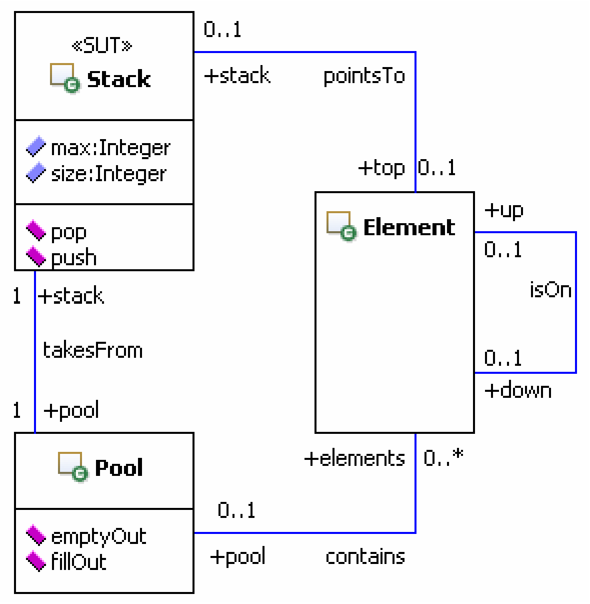

Figure 1. The class diagram of the Stack model

The *Stack::push()* and *Stack::pop()* operations are events used in the state-machine.
Pool::emptyOut() and *Pool::fillOut()* are defined as follows:

context: Pool::emptyOut():OclVoid
post: self.elements-> isEmpty()
  /*@REQ: pool_empty@*/

context: Pool::fillOut() : OclVoid
post: selt.elements = Element.allInstances()
  /*@REQ: pool_fill@*/

Notice the use of the *isEmpty()* operator in an active context in order to empty out the pool. Also notice the two requirements set on these operations: *pool_empty* and *pool_fill*.

3.2.2 Initial state

Figure 2 presents the object diagram that depicts the initial state of the system under test.

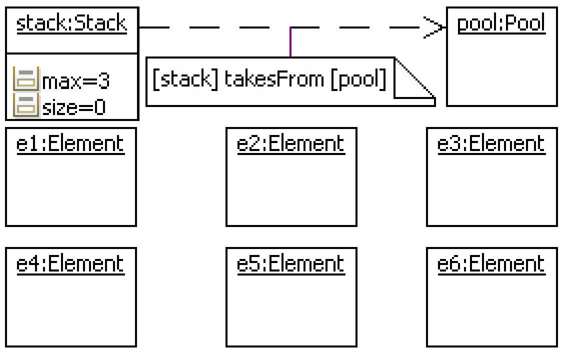

Figure 2. The initial state of the Stack model

In the initial state, the stack and the pool are empty. The link between the stack and the pool is created. Note the MAX constant, arbitrarily set to 3.

3.2.3 State-machine

Figure 3 presents the state-machine used to describe the different dynamic states of the system under test.

The state-machine is clearly comprehensive. We can push and pop elements. The different states in which the stack can be are designed in this diagram. The transition actions are defined as follows:

```
action pushOnEmptyStack
post:
   let element = self.pool.elements->any(true) in
                    self.top = element
                    /*@REQ:random_element@*/
                    and self.size = self.size + 1
                    and self.pool.elements =
                    self.pool.elements->excluding(element)
                    /*@REQ:automatic_delete@*/


action pushOnLoadedStack
post:
   let element = self.pool.elements->any(true) in
                    element.down = self.top
                    /*@REQ:random_element@*/
                    and self.top = element
                    and self.size = self.size + 1
                    and self.pool.elements =
                    self.pool.elements->excluding(element)
                    /*@REQ:automatic_delete@*/

action popForEmptyStack
post:
  let element = self.top in
                    self.top.oclIsUndefined()
                    and self.size = self.size - 1
                    and self.pool.elements =
                    self.pool.elements->including(element)
                    /*@REQ:automatic_reinsertion@*/


action popForLoadedStack
post:
  let element = self.top in
                    self.top = element.down
                    and element.down.oclIsUndefined()
                    and self.size = self.size - 1
                    and self.pool.elements =
                    self.pool.elements->including(element)
                    /*@REQ:automatic_ reinsertion @*/
```

Notice the particular use of active expression expressed with operators that are interpreted as mentioned in 2.2.2 (*oclIsUndefined()*). The two requirements *random_element* and *automatic_delete* are linked with the actions *push**. The requirement *automatic_reinsertion* is set on the actions *pop**. The requirements *empty_stack_exception* and *full_stack_exception* are both linked to empty actions, written with the keyword *true.*

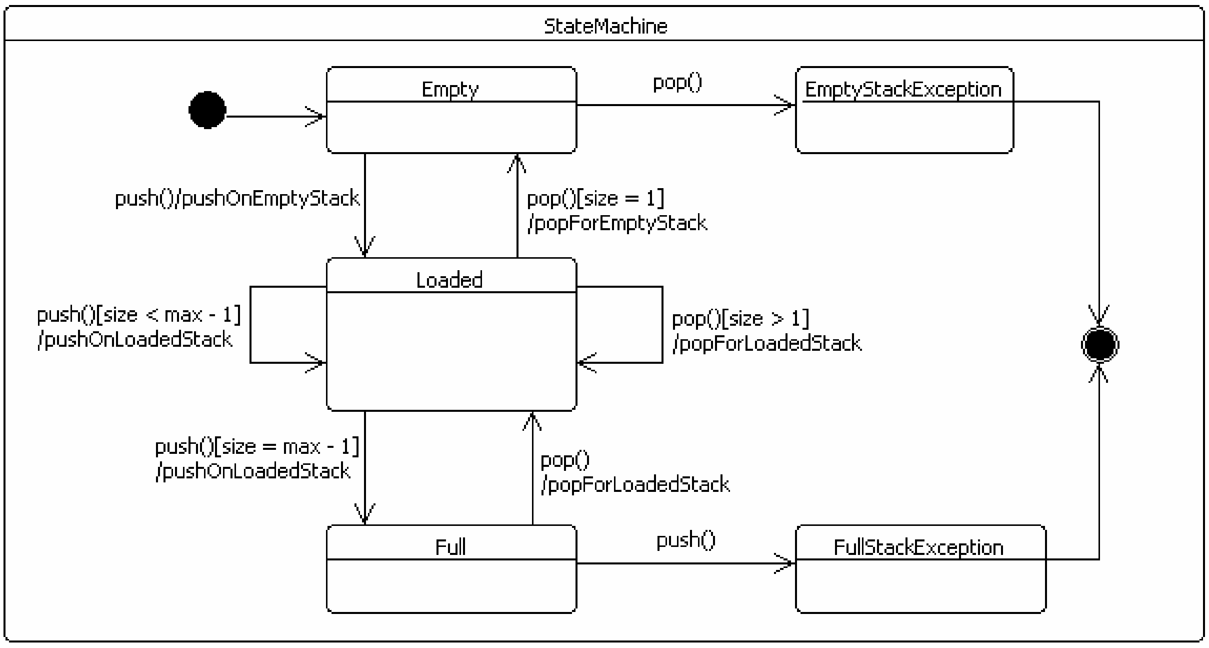

Figure 3. The state-machine of the Stack model

3.3 Test targets and generated tests

Table 11. More precisely, for each test target, we use an automated theorem prover [8] to search for a path from the initial state to that target, and data values that satisfy all the constraints along that path. This is similar to a symbolic model-checking approach [9, 10]. A test is also composed of:
• a *preamble* (potentially empty); the sequence of operations called to reach the targeted behaviour,
• a *body*, the execution of the targeted behaviour,
• a *postamble* (potentially empty); the sequence of operations to return to the model initial state.
The generation of postambles is optional.
In the Stack model, the generated tests cover all the behaviours that were modelled, and all the states and transitions of the state-machine.
In addition we can construct the traceability matrix of the requirements that are designed in the model and linked with the test targets.
Note that some tests have no postamble. This means that the model initial state is not reachable from the state, in which the system under test is.

Table 10. Test targets from the Stack model

Id	Tested UML element	Targer deinition		Tested Requirements
		context	effect	
Operations				
1	*POOL::emptyOut*	-	elements->isEmpty()	pool_empty
2	*POOL::fillOut*	-	elements=Element.allInstances()	pool_fill
Transitions				
3	*Empty EmptyStackException*	-	true	empty_stack_exception
4	*Empty Loaded*	-	let element = pool.elements->any(true) in top=element and size=size+1 and pool.elements->excludes(element)	random_element, automatic_delete
5	*Loaded Loaded*	size < max-1	let element = pool.elements->any(true) in element.down=top and top=element and size=size+1 and pool.elements->excludes(element)	random_element, automatic_delete
6	*Loaded Full*	size = max-1	let element = pool.elements->any(true) in element.down=top and top=element and size=size+1 and pool.elements->excludes(element)	random_element, automatic_delete
7	*Full FullStackException*	-	true	full_stack_exception
8	*Full Loaded*	-	let element = top in top=element.down and element.down.oclIsUndefined() and size=size-1 and pool.elements->includes(element)	automatic_reinsertion
9	*Loaded Loaded*	size > 1	let element = top in top=element.down and element.down.oclIsUndefined() and size=size-1 and pool.elements->includes(element)	automatic_reinsertion
10	*Loaded Empty*	size = 1	let element = top in top.oclIsUndefined() and size=size-1 and pool.elements->includes(element)	automatic_reinsertion

Table 11. Generated tests on Stack model

Target Id	Corresponding test		
	Preamble	Body	postamble
1		pool.emptyOut()	
2		pool.fillOut()	pool.emptyOut()
3		stack.pop()	
4	pool.fillOut()	stack.push()	stack.pop(), pool.emptyOut()
5	pool.fillOut(), stack.push()	stack.push()	stack.pop(), stack.pop(), pool.emptyOut()
6	pool.fillOut(), stack.push(), stack.push()	stack.push()	stack.pop(), tack.pop(), stack.pop(), pool.emptyOut()
7	pool.fillOut(), stack.push(), stack.push(), stack.push()	stack.push()	
8	pool.fillOut(), stack.push(), stack.push(), stack.push()	stack.pop()	stack.pop(), stack.pop(), pool.empty()
9	pool.fillOut(), stack.push(), stack.push()	stack.pop()	stack.pop(), pool.empty()
10	pool.fillOut(), stack.push()	stack.pop()	pool.empty()

4. RELATED WORK

They are numerous model-based testing approaches that use UML as modelling notation[4]. Some of them are based on sequence or interaction diagrams to express scenarios (see e.g. [11]), state machines to express behaviour models (see e.g. [12]) or combine them (see e.g. [13]). Few approaches are using OCL as an action language for model-based testing. B. K. Aichernig proposes an approach based on mutation analysis of OCL specifications [14], and Bruel et al [15] proposes a combination of test cases using an approach very similar to the test target computation proposed in this paper. But there is currently no subset of UML/OCL clearly proposed for model-based testing.

5. CONCLUSION

This paper introduced a subset of UML/OCL for model-based testing. In Section 3, we illustrated how this UML-MBT subset of UML can be used to write a precise model of a Stack system, which is executable and a good basis for test generation. The stack model is very small, and it would not be difficult to generate a similar test suite manually – but with larger industrial models many more tests are needed to cover the model, and the cost benefits of model-based testing become more significant. The UML-MBT subset of UML is fully supported by the LEIRIOS Test Designer v3.0 tool (see [3, 4] for more detail on test generation strategies). This test generator takes UML models from Borland Together and IBM Rational Software Modelling tools and provides a plug-in that verifies the compliance of the UML model with the defined UML-MBT subset. It checks the model for OCL

[4] See "model-based testing" section on Wikipedia to have an updated list of MBT tools - http://en.wikipedia.org/wiki/Model-based_testing

syntactic verification and consistency (e.g. verification that the instances verify the corresponding multiplicities in the class diagram). LEIRIOS Test Designer provides adapters to export generated test cases and test scripts in test management and execution tools such as HP/Mercury Quality Center. This UML-based model-based testing solution is currently deployed on large applications in the domains of Enterprise IT information systems and eTransactions systems (banking, ticketing or e-Admin applications).

6. REFERENCES

[1] S. J. Prowell, *"JUMBL: A Tool for Model-Based Statistical Testing,"* hicss, p. 337c, 36th Annual Hawaii International Conference on System Sciences (HICSS'03) - Track 9, 2003.

[2] M. Utting and B. Legeard. *Practical Model-Based Testing - A tools approach.* Elsevier Science/Morgan&Kaufmann, 2007. 454 pages, ISBN 0-12-372501-1.

[3] E. Bernard, F. Bouquet, A. Charbonnier, B. Legeard, F. Peureux, M. Utting, and E. Torreborre. "Model-based Testing from UML Models". In *Procs. of the Int. Workshop on Model-based Testing (MBT'2006)*, volume P-94 of *Lecture Notes in Informatics*, Dresden, Germany, pages 223-230, October 2006. ISBN 978-3-88579-188-1.

[4] F. Bouquet, E. Jaffuel, B. Legeard, F. Peureux, and M. Utting. "Requirement Traceability in Automated Test Generation - Application to Smart Card Software Validation". In *Procs. of the ICSE Int. Workshop on Advances in Model-Based Software Testing (A-MOST'05)*, St. Louis, USA, May 2005. ACM Press.

[5] J. Warmer and A. Kleppe. The Object Constraint Language Second Edition: Getting Your Models Ready for MDA. Addison-Wesley, 2003.

[6] G. J. Myers. *The Art of Software Testing*. John Wiley & Sons, 1979.

[7] A.J. Offut, Y. Xiong and S. Liu. "Criteria for generating specification-based tests", Proceedings of the 5th Int. Conference on Engineering of Complex Computer Systems (ICECCS'99), Las-Vegas, USA, pages 119-131, October 1999. IEEE Computer Society Press.

[8] Prover Technology website – www.prover.com

[9] T. Jéron and P. Morel, "Test generation derived from model-checking", Proceedings of the 11th Conference on Computer-Aided Verification (CAV'99), Trento, Italy, LNCS 1633, pages 108-122, July 1999

[10] P. Ammann, P.E. Black and W. Majurski, "Using Model Checking to Generate Tests from Specifications", Proceedings of the 2nd Int. Conference on Formal Engineering Methods (ICFEM'98), Brisbane, Australia, pages 46-55, December 1998. IEEE Computer Society Press.

[11] Matthias Beyer, Winfried Dulz, Fenhua Zhen, "Automated TTCN-3 Test Case Generation by Means of UML Sequence Diagrams and Markov Chains," *ats*, p. 102, 12th Asian Test Symposium (ATS'03), 2003.

[12] M.E. Vieira, M.S. Dias, D.J. Richardson, Object-Oriented Specification-Based Testing Using UML State-chart Diagrams, Proceedings of the Workshop on Automated Program Analysis, Testing, and Verification (at ICSE'00), June 2000.

[13] L. Briand, Y. Labiche, A UML-Based Approach to System Testing, Proceedings of the Fourth International Conference on the Unified Modeling Language (UML'01), 2001, pp. 194-208.

[14] Bernhard K. Aichernig, Percy Antonio Pari Salas: Test Case Generation by OCL Mutation and Constraint Solving. QSIC 2005: 64-71

[15] M. Benattou, J.-M. Bruel, and N. Hameurlain, “Generating Test Data from OCL Specification” in Proceedings of the ECOOP’2002 Workshop on Integration and Transformation of UML models (WITUML’02), 2002.