UML-CASTING: Test synthesis from UML models using constraint resolution[1]

Lionel Van Aertryck[1] and Thomas Jensen[2]

[1] Alliance Qualité Logiciel - Groupe Silicomp,
Rue de la Chataîgneraie, BP 127
35513 CESSON SEVIGNE CEDEX, FRANCE
Lionel.Van.Aertryck@aql.fr
[2] IRISA / CNRS
Campus Universitaire de Beaulieu
Avenue du Général Leclerc
35042 RENNES CEDEX, FRANCE
Thomas.Jensen@irisa.fr

Abstract. We present UML-CASTING, a prototype tool for computer-assisted generation of test suites from UML models. The concept underlying Casting is that a large part of test cases corresponding to a given test strategy can be obtained automatically from the text of the specification. A test strategy consists of a set of decomposition rules that are applied to the expressions in a specification in order to reveal potentially interesting test cases. In the case of UML-CASTING, the decomposition is applied to class and state diagrams in order to yield an intermediate representation (also in the shape of a state machine) from which test (here, method invocations) can be extracted. Expressions are written in a subset of OCL[2]. The user can then specify two kinds of goals: either a coverage percentage that he wants to achieve or specific sequence of method calls that he wants to see executed. In either case, UML-CASTING will use constraint-solving techniques to produce test cases to achieve these goals.

1. Introduction

Testing is a key component in the validation of software. It is an activity that can represent up to 40-50% of the total development cost of a piece of software [5] and there is as a consequence a major industrial demand for methods and tools that can automate this process. In this paper, we present an automatic method for generating test data from UML specifications [21] according to a user-defined test strategy.

Our method is based on the well-known testing technique that consists in dividing data into equivalence classes on which a program is considered to behave identically. For example, we might want to execute a comparison $x \geq y$ in the following three cases $x < y$, $x = y$ and $x > y$ which then define three equivalence classes for this expression. In general, there are two major difficulties associated with this approach: reasonable equivalence classes have to be defined and representatives of these classes have to be identified. Here, we propose a systematic and pragmatic approach to solve these problems aiming at automating what can be automated. The idea underlying our approach is that equivalence classes are implicitly expressed in semi-formal models (here UML specifications) and have to be revealed.

A test strategy defines the equivalence classes for each basic operator (logical and arithmetic) in the language. Applying such a strategy to a specification identifies a collection of atomic test transitions (execute $x \geq 0$ with $x = 0$ and $x > 0$) that should be executed. Once these test transitions have been identified, the problem becomes how to bring the program into a state in which the transitions can be performed. To solve this problem, the specification is combined with constraint-solving techniques in order to trace back a path to the initial state of the specification.

Two kinds of interaction are possible with the tool. First use of the tool is the automatic generation of test cases to meet a given coverage criteria. Second use is to generate test cases in a semi-automatic way from test patterns identified by the user.

CASTING is a general method for test case generation from formal or semi-formal models that can be instantiated to different input formalisms. The underlying techniques of CASTING have been presented in [3]. Initial works performed to instantiate CASTING have been done for a subset of AMN (the B-Method language). The method relies on attribute grammars to analyze the model and on constraint-solving techniques to search for a path and for test data selection. In the context of the French RNTL project [9], we have developed the

[1] This work has been supported by RNTL project COTE.
[2] The Object Constraint Language OCL allows to specify constraints on parts of an UML model.

instantiation of CASTING to UML. This article summarizes first results obtained during this project, including the presentation of a prototype implementing the instantiation of CASTING to UML. The goal of the work presented in this paper was to determine if CASTING techniques can be applied to UML, and it was not to cover UML. Consequently, we have made restrictions on the input model to cope with a minimal starting point, close to what has already been done for the subset of B. This work is a first step toward a future and more complete instantiation of CASTING to UML.

The remainder of this paper is organized as follows. Sections 2 to 0 provide detailed information about the principles and techniques underlying UML-CASTING and its application to a UML model. Section 3 presents the application of the method to an example of a small model of a banking account. Related works. i.e. test cases generation using Constraint Logic Programming (CLP) techniques are presented in Section 4. We present shortly the prototype in section 5. Section 6 outlines future research directions.

2. UML-CASTING

Principles of UML-CASTING

Figure 1 presents a general view of the method and an overview of its interfaces. The method takes as input a model satisfying several constraints and a test strategy to build an intermediate representation that will be used to generate test sequences. This internal representation is in the shape of a finite state automaton that contains exactly the same states as in the state diagram but where each transition of the state diagram is replaced with a set of transitions. These transitions are specific reduction of the input domain of the initial transitions. This representation of the model is the result of the application of a selection of decomposition rules to the logical expressions occurring in the model and expressed in a formal language. This intermediate representation is then used either to generate test sequences (methods call with valued parameters) or to check and complete test patterns submitted by the user. For both uses, a test coverage measure is computed and provides a feedback to the user on the adequacy of the test cases and the chosen test strategy.

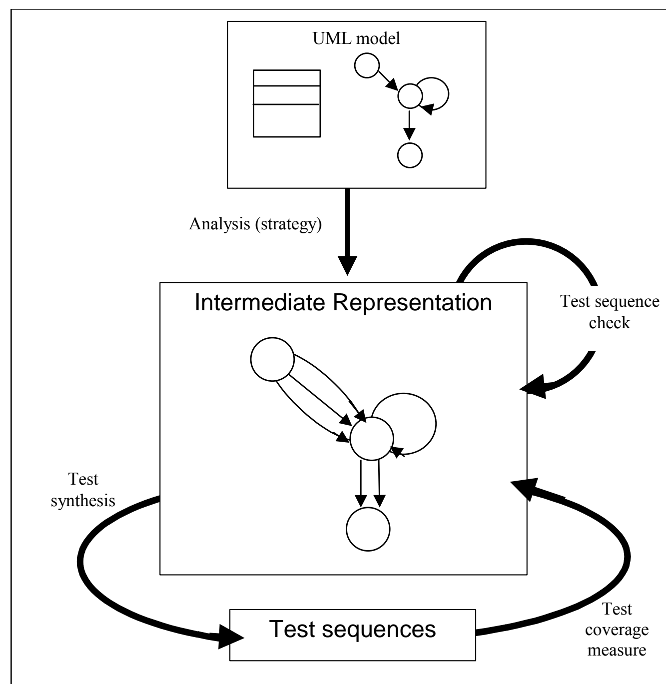

Figure 1: UML-CASTING interfaces

In the following sections (0 to 0) we describe all the steps that compose the test generation process.

Input Model

In this section we detail the part of UML that is taken into consideration. The model must have behavior information for the technique to produce interesting test cases. This information is primarily present in the state diagram but can be complemented by the precondition and postcondition of the class diagram. Obviously, the more the user defines the model the more detailed the test cases will be (this depends also on the test strategy, though).

We have made several strong restrictions on UML models to get an overview of the feasible instantiation of CASTING to UML. The model is made of a single class diagram and a single state diagram. Consequently, the tests generated are for a single object. OCL expressions are limited to the grammar presented in Figure 2.

The class diagram and the state diagram are the sources of input for the application of UML-CASTING. Concerning the class diagram, type of attributes, possible invariant expression constraining the attributes and method definition (signature, precondition and postcondition) are exploited to build the intermediate representation. The state diagram, if available, contains also useful information. Indeed, the state diagram contains control flow information that is of paramount importance to limit the constraint resolutions steps during the path exploration. If the state diagram is not defined, a virtual one is supposed to exist that contains only one init state and allows any method to be applied with the only constraint that the precondition (from the class diagram) of the method triggered by the transition is satisfied.

The intermediate representation of the model includes control flow information that is used during the search for valid test sequences, in conjunction with constraints obtained from the logical expressions analysis.

Logical expressions are expressed using a subset of OCL (Object Constraint Language) [18]. As explained earlier, logical expressions occurring in several structural elements of the model are subject to decomposition rules. In order to stay as close as possible to UML, a subset of OCL is used to express these logical expression in the model. This subset, presented in Figure 2, contains logical and arithmetic operators. This subset does not contain operators and operations on collections (forall, exists, iterate, collect, ...), String, Real, enumeration types, let expressions, function calls and associations. The type of a variable is limited to either Integer or Boolean. Postcondition, either in the class diagram or in the state diagram can refer to previous state variables (using the OCL notation @pre). Thus, it is possible using the postcondition to modify the system states implied by transition triggering. The formal language OCL (Object Constraint Language) has been specifically designed for UML to allow the model designer to quickly and easily express properties in a rigorous way without having to learn a new and possibly complex formal language. The use of such a language in a specification task offers several well-known improvements like avoiding ambiguities and the possibility to conduct automated analysis. The latter is of prior interest for our approach to tests generation. Examples of expression conforming to this grammar can be found in Section 3 dedicated to the example.

```
expression                 :=  logicalExpression
ifExpression               :=  "if" expression "then" expression "else" expression "endif"
logicalExpression          :=  relationalExpression (logicalOperator relationalExpression )*
relationalExpression       :=  additiveExpression (relationalOperator additiveExpression) ?
additiveExpression         :=  multiplicativeExpression (addOperator multiplicativeExpression ) ?
multiplicativeExpression   :=  unaryExpression (multiplyOperator unaryExpression )*
unaryExpression            :=  (unaryOperator postfixExpression ) | postfixExpression
postfixExpression          :=  literal | "(" expression ")" | propertyCall | ifExpression
propertyCall               :=  pathName (timeExpression ) ?
pathName                   :=  name
literal                    :=  string | number | booleanConstant
booleanConstant            :=  "TRUE" | "FALSE"
timeExpression             :=  "@pre"
logicalOperator            :=  "and" | "or" | "xor"
relationalOperator         :=  "=" | ">" | "<" | ">=" | "<=" | "<>"
addOperator                :=  "+" | "-"
multiplyOperator           :=  "*" | "/"
unaryOperator              :=  "not"
```

Figure 2: Subset of OCL.
(**x**)? means optional and (**x**)* means 0 or more.

Test strategy

Defining a test strategy is like choosing elementary testing techniques in a database and combining them. The user must have a global test intention before starting the elementary testing techniques selection. Elementary

testing techniques are expressed with what we call decomposition rules. Rules are either structure decomposition rules or operator decomposition rules. Structure decomposition rules indicates if a structure element of a model like precondition of method, guard condition for transition in the state diagram, invariant for the class diagram have to be taken into account in the process, whereas operator decomposition rules apply to expressions located in these structure elements: text of a precondition, text of a postcondition, type of an attribute, etc....

Concerning the structured elements, the decomposition rules are limited to "decompose" or "do not decompose". If "decompose" is chosen, then the logical expression is analyzed and decomposition rules for operators are applied and used to identify test cases. Figure 3 presents in a graphical way the structure decomposition rules and the structured element of the specification they are related to.

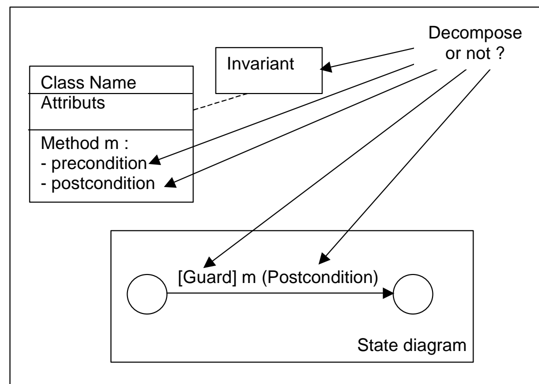

Figure 3: Structure decomposition rules

Concerning the decomposition rules for the operators, it is possible to choose between several possibilities of decomposition for each operator. As shown in Figure 4, each rule contains several decomposition techniques (rule1, rule2, etc..), without any mandatory hierarchy between them. These rules are not defined by the user but are presented to the user as choices offered by the tool. First rule corresponds to the case where no subdomain has to be made explicit accordingly to the operator use. The others rules are different ways of identifying subdomains known, by practice, to be good candidates for test data selection. One important point here is that rules are not defined only for comparison operators but also for computation operators (like addition in Figure 4). This kind of rule allows to generate a test for the case where the result of the addition is zero for example (rule3 in Figure 4 for + operator).

Rule for A + B	**Rule for** A >= B:
Rule1: no decomposition	**Rule1:** no decomposition
Rule2: (A = 0 or A <> 0) and (B = 0 or B <> 0)	**Rule2:** (A > B) or (A = B)
Rule3: (A = -B) or (A <> -B)	**Rule3:** (A = B) or (A = B+1) or (A > B+1)

Figure 4: Examples of operator decomposition rules

Each case defined in a decomposition rule makes explicit one or several specifics domains that have to be covered by the final decomposed expression. It is up to the rule designer to define case that will not lead to useless generated test. For example, it would have been useless to add a constraint A < B for >= , because the application of that rule will produce the case (A >= B and A < B) which is obviously an unsatisfiable expression.

The underlying technique for the implementation of both kinds of decomposition rules is attributed grammars. Using attributed grammars it is possible to compute and extract information from a syntactic analysis of a textual representation of the model. The test strategy is transmitted through an inherited attribute whereas synthesis attributes are used for the translation of the OCL expressions into our internal language (i.e. constraint expressions) and the subdomains identified by the application of the decomposition rules. More details concerning this part of the method can be found in [3].

In practice, the user does not have to choose which rule will be applied for each operator. Indeed several already defined test strategies, gathering a selection of rules for each operator, are proposed. The user still has the possibility to fine tune the strategy using the structure decomposition rules. This will select the elements of the model to which the user wants the decomposition rules for the operators to be applied: invariant, precondition and postcondition in the class diagram, guard and postcondition in the state diagram.

The second level of testing, the basis test strategy, consists in limiting decomposition to OCL expressions that are in the guard and the precondition of the transition. By applying this strategy to a model, the user obtains a set of test transitions that makes explicit subcases in the input domain.

The decomposition rules always cover the whole input domain through the identified test transitions, even if they do not have to generate disjoint partitions of an application domain. This is necessary to insure that the resulting test transitions will together completely cover the initial input domain and that no model behavior will be “forgotten” during the test generation process due to badly defined decomposition rules.

Intermediate representation

The intermediate representation of the model is the result of several transformations and computations on the initial model. First, a comprehensive textual representation of the model is produced from the UML model. This representation contains all the information in text format that will be taken as input for the method. Then this representation is analyzed and the decomposition rules are applied leading to the definition of the intermediate representation in the shape of a state diagram. Transitions of the intermediate representation are test transitions resulting from the application of the decomposition rules. This intermediate representation will then be used to identify and select valid sequences of methods call that achieve a given coverage measure.

One particularity is that the states in the intermediate representation are the same as in the model ; i.e., there is no new state defined. The tool does not deal with hierarchical state diagrams in its actual version but could be combined with transformation techniques such as those described by Kim *et al.* [16] to remedy this problem. Test transitions are obtained by merging information extracted from the class diagram and from the state diagram into a single expression that is then put into disjunctive normal form.

The input of the decomposition is a transition in the state diagram. There are constraints and information linked to every transition: the method call associated with the transition with possibly a precondition and a postcondition, a guard and a postcondition for the transition. Moreover, an invariant may have been defined on the attributes in the class diagram and should be taken into account.

The following example, taken from the example fully presented in section 3, shows how a set of test transitions is produced from a UML model with a given test strategy.

The following information is extracted from the class diagram and the state diagram of the model.

public deposit (x in integer, result out integer) **pre**: x > 0 **post**: result = sum **invariant**: sum >= 0 and sum <= 100	*Class diagram Information*
guard of transition: true **post of transition**: sum = sum@pre+x	*State diagram Information*

The test strategy[3] consists in the decomposition of all the elements of the model. The decomposition rules for the operators **>=** and <= are:

A >= B is decomposed into (A = B or A > B). **(1)**

A <= B is not decomposed. **(2)**

[3] This test strategy is only for this example. The test strategy in section 3 is different and the transition 7 will be decomposed in 4 test transitions rather than only 2.

The rough result of the application of the test strategy (i.e. the decomposition rules) to the transition deposit in the state unlocked is the formula F which is a made of the conjunction of the pre and postcondition, the guard and postcondition of the transition, the class invariant and the OCL expressions produced by the application of the decomposition rules:

F = sum@pre <= 100 and (sum@pre > 0 or sum@pre = 0)
and sum <= 100 and (sum = 0 or sum > 0)
and x > 0 and sum = sum@pre + x and result = sum **(3)**

This formula is then transformed into disjunctive normal form in order to make explicit each equivalence class:

DNF(F) = case1 or case2 or case3 or case4 **(4)**

with

sum@pre > 0 and sum@pre <= 100 and sum > 0 and sum <= 100 and x > 0 and sum = sum@pre + x and result = sum	*case1*
sum@pre = 0 and sum@pre <= 100 and sum > 0 and sum <= 100 and x > 0 and sum = sum@pre + x and result = sum	*case2*
sum@pre > 0 and sum@pre <= 100 and **sum = 0** and sum <= 100 and **x > 0** and **sum = sum@pre + x** and result = sum	*case3*
sum@pre = 0 and sum@pre <= 100 and **sum = 0** and sum <= 100 and **x > 0** and **sum = sum@pre + x** and result = sum	*case4*

Refining and exploiting the generated tests

As the test transition generation process is syntactic in its first stage, the result may contains transitions with unsatisfiable constraints. Consequently, a second stage aiming at eliminating a large part of these useless transitions is performed that mainly relies on constraint resolution. Concerning the example, cases 3 and 4 are obviously not satisfiable. Case 3 and case 4 contain contradiction: the account (sum) cannot be empty after the addition of a positive quantity: x>0. These cases should not be kept for the following stages of the test transitions identification. Consequently, once a constraint resolution for the case definition concludes that no solution exits (i.e. a contradiction has been found and the domain of at least one variable is empty), the case is withdrawn.

On the other hand, even if cases 1 and 2 are satisfiable, this does not imply that it is possible to build a test sequences that includes them from the init state to a state satisfyng their preconditions. Indeed, there may be no path going from the initial state to a state satisfying their constraints. Suppose for example that the attribute sum is equal to 100 at the creation (due to a postcondition in the creationAccount method for example), then case 1 will never be applied because in that case the attribute sum of the resulting state will not satisfy the invariant. This case will be not reachable and the tool will not be able to exhibit a test sequence containing it. In order for the tool to be able to explain why the test transition has not been covered, it requires to explore the entire domain for all the methods parameters and attributes, which is a very time-consuming process. Even if more sophisticated constraint-solving techniques exist, a constraint solver generally propagates the effects of the constraints on the variables until it reach a point where no more constraint can be propagate. At this point, and depending on the resolution algorithm, a variable is selected among the ones that are not instantiated and a value from the possible ones is tried. The effect of this choice is then propagated to other variables until a solution is found or all the possibilities have been tried.

Coming back to the example, two test transitions (case 1 and case 2) are identified and proposed to the user at the end of the intermediate representation building. Their integration in the automaton is done by replacing the transition deposit in state unlocked in the states diagram by two generated transitions, one for each case. By construction these transitions gather the invariant, the precondition and postcondition of the method and also the guard and the postcondition of the transition with additional constraints obtained by decomposition. These constraints define the domain reduction associated to the equivalence class.

At this stage, the user has several ways to interact with the tool. First, an analysis of the test transitions produced can reveal problems in the model. Suppose for instance that the user has a specific case of application in mind for a method and that none of the test transitions for that method covers it. This can reveals that the model is not a correct specification of the user intention. The user has to determine the origin of the difference: a

too constraining invariant, an incomplete set of guards, ... Second, the user can modify the test strategy before continuing in the test synthesis process, depending on the level of detail of the test transitions produced with respect to his expectation. Finally, the user can use the generated test transitions to evaluate a set of already defined test sequences, including test sequences not completely valued. The tool will then confront the user test sequences with the internal representation and provides as a result a verdict of conformance for the test sequence. Moreover, and depending of this verdict, an instantiation of the method parameters occurring in the sequences can be provided.

In summary, the intermediate transition can be viewed as a refinement of the initial state diagram where each transition is replaced by a set of transitions (test transitions) distinct from each other with more restrictive guards. The transition can be pruned by automatic and manual means to produce a final set of tests.

Test sequence generation

This section presents the principles of the test sequence generation. The test sequence generation aims at covering the test transition identified in previous stage through exploration of the intermediate representation and selection of a set of paths. This exploration is being made possible by the representation based on constraints of the test transitions. The intermediate representation is built once and can be used several times to produce test sequences depending on different test transition coverage rate. The intermediate representation can also be used to check and complete user defined test sequences.

The algorithm for test sequence generation explores the intermediate representation from the initial state with a breadth-first search and tries to apply each test transition at least once. A test transition is selected and is considered covered if its associated constraint is consistent with the constraint system corresponding to the previous method calls on the path. The constraint resolution is partial during this step, it doesn't state the existence of a solution but it prunes the domains of possible values for each variable. The search for a set of values is done later in the process. In order to achieve a given coverage rate of test transitions, priority is given to the transitions not already covered whenever it is possible. Once the test sequence stops, i.e. ideally all the test transitions have been covered, the ultimate constraint resolution is performed in order to choose a value for all the method parameters. At the end, the user gets a set of sequences of valued method calls. The data selection is based on the hypothesis that a test performed with a data representing a subdomain, i.e. as defined by a test transition, is equivalent w.r.t. the model behavior to the others and thus is representative of the whole subdomain defined by the test transition. We do not consider additional subdomains introduced by previous transitions along the test sequence. If a test succeeds, i.e. the result is the expected one, then the subdomain is considered as being tested even if the test is done for only one value of the subdomain defined by the test transition. This corresponds to the uniformity hypothesis defined in [6], which is a formal expression of a common hypothesis but not always being made explicit in the equivalence classes testing techniques.

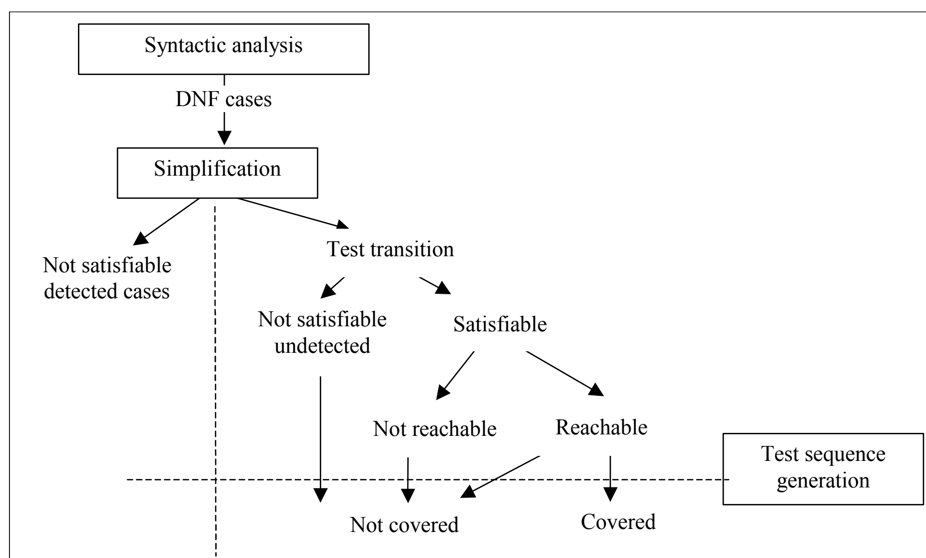

Figure 5: Different uses and kind of test transitions

Figure 5 presents the different kinds of test transitions. As shown by this classification, it has to be noted that the algorithm cannot always insure a 100% coverage of the test transitions, even in a reasonable amount of time. This may come from the fact that the resolution takes too long or because the tool tries to cover test

transitions that are unreachable in practice. At the end of the test sequence process, it is up to the user to determine the kind of each uncovered test transition: reachable but not covered due to generation limitations, not covered because not reachable from the initial state, not covered because unsatisfiable but not eliminated by the simplification. In such a case, the user has to help the tool to complete the automatically produced tests. Defining a test sequence pattern that will be submitted to the tool can do this. As a result, the tool provides an instantiated version of the test sequence with a coverage measure of the test transitions. This approach lets the user focus on difficult test cases design instead of writing a large quantities of test cases, which can be a tedious and borrowing task.

There are a number of user-adjustable parameters that allow to control the test generation process. This is necessary due to the fact that the test coverage objective may not be achievable within a finite (or even reasonable) amount of time. Consequently, in practice the user has to limit the depth of the exploration of the intermediate representation and has to chose an acceptable coverage rate before launching the test suite generation.

3. Example: a banking account

In this section we demonstrate the test production chain on a UML specification of a bank account. The model contains two attributes and four operations. The state diagram contains 9 transitions. The resulting intermediate representation will contain 28 test transitions. A sequence of length 28 covering all of them will be automatically produced.

The model

The model is a simple bank account that can be locked or unlocked. Two methods are used to deposit or withdraw money. One additional method is used to set a lock on the account. The amount of money available cannot be less than zero or greater than 100. No deposit or withdrawal is possible when the account is locked. It is possible to withdraw only a positive value. It is not possible to withdraw more than what is available in the account.

Attributes :
public boolean locked
public integer sum
Invariant for the class BankingAccount:
invariant: sum >= 0 and sum <= 100
Operations :
public setLock (lock in boolean):
pre: true **post**: locked = lock
public withdraw (x in integer, result out integer):
pre: x > 0 and x <= sum@pre **post**: result = sum
public deposit (x in integer, result out integer):
pre: x > 0 **post**: result = sum
public createAccount ( ):
pre: true **post**: locked = FALSE and sum = 0

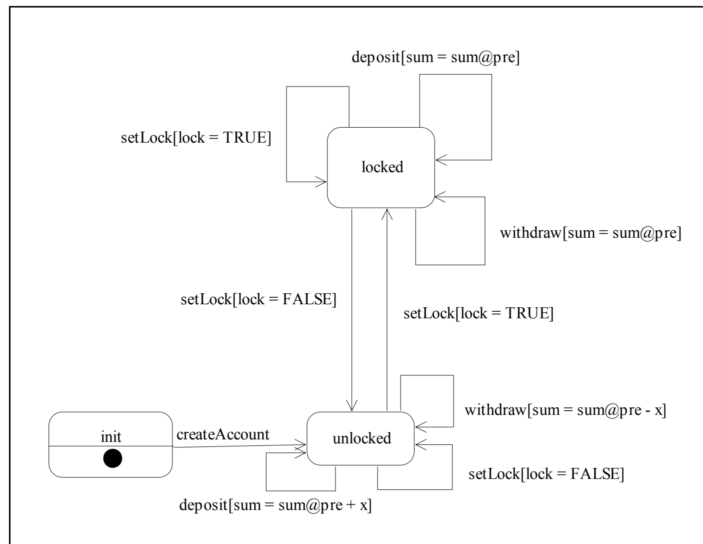

Figure 6: State diagram
The UML syntax for transition is: event-name (parameters) [postcondition]

The test strategy

The test strategy consists in decomposing all the elements of the model and all the arithmetic comparison operators.

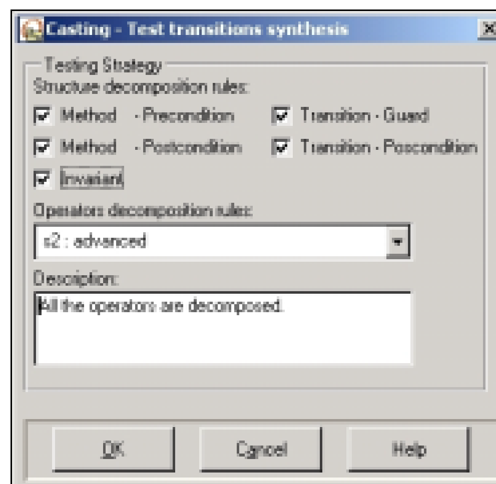

Figure 7: Test strategy interface

Intermediate representation

Once the strategy is chosen, it is applied to the model in order to build the intermediate representation. We do not present here a formal definition of the intermediate representation because this definition can be found in previous papers about CASTING ([1] and [2]). To summarize, the intermediate representation is a state diagram composed of the same states as in the original UML state diagram where transitions between states are replaced with the test transitions generated and expressed using a constraint language.

Test transitions are generated and gathered into a single state diagram.

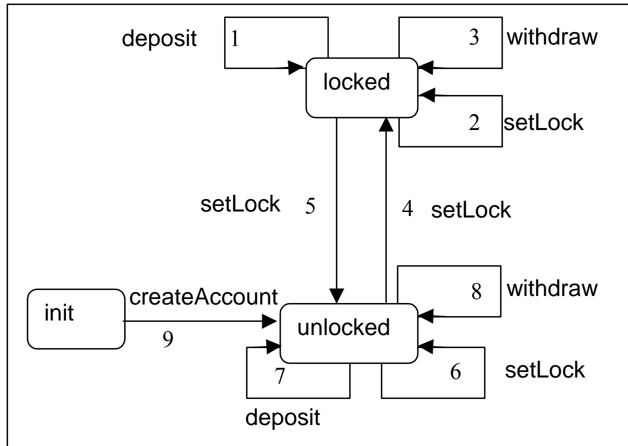

Figure 8: State diagram with labeled transitions

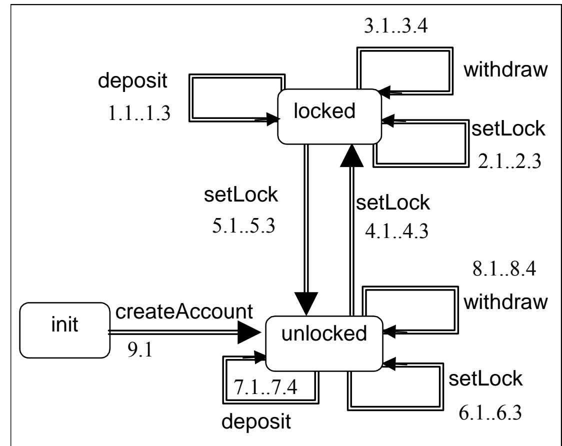

Figure 9: Intermediate representation

Each transition in the state diagram is labeled (Figure 8) and replaced by the associated generated test transitions, leading to the building of the intermediate representation (Figure 9). For example, transition 5-SetLock is replaced by test transitions 5.1, 5.2 and 5.3 (represented with a double line arrow) in the intermediate representation.

The test transitions

The decomposition of the transitions produces from 1 to 4 test transitions per transition. As an example, 8 cases have been identified for the method withdraw (4 for each input state: transition number 3 and transition number 8). We present them in Table 1, with an informal description. The automated and systematic generation provides a comprehensive set of test transitions, resulting from a coherent and complete application of a given test strategy to the whole model.

Test transition	Constraint	Informal description

3.1
locked to locked
	`sum@pre < 100 and sum@pre > 0`
	`and sum < 100 and sum > 0`
	`and x < sum@pre and x > 0 and sum = sum@pre`
		The account is locked, it is neither full nor empty and a withdrawal is done that does not have any impact.

3.2 locked to locked	`sum@pre = 100 and sum = 100` `and x < sum@pre and sum = sum@pre and x > 0`	The account is locked and full and a withdrawal is done that does not have any impact.

3.3
locked to locked
	`sum@pre > 0 and sum@pre < 100`
	`and sum > 0 and sum < 100`
	`and x = sum@pre and x > 0 and sum = sum@pre`
		The account is locked, it is neither full nor empty and a withdrawal is done of exactly the amount, but has not any impact.

3.4 locked to locked	`sum@pre = 100 and sum = 100` `and x = sum@pre and x > 0 and sum = sum@pre`	The account is locked and full and a withdrawal is done of exactly the amount, but has not any impact.

8.1
unlocked to unlocked
	`sum@pre < 100 and sum@pre > 0`
	`and sum < 100 and sum > 0`
	`and x < sum@pre and x > 0 and sum = sum@pre-x`
		The account is unlocked, it is neither full nor empty and a withdrawal is done that does not empty the account.

8.2
unlocked to unlocked
	`sum@pre = 100`
	`and sum < 100 and sum > 0`
	`and x < sum@pre and x > 0 and sum = sum@pre-x`
		The account in unlocked and full and a withdrawal is done that does not empty the account.

8.3
unlocked to unlocked
	`sum@pre < 100 and sum@pre > 0`
	`and sum = 0`
	`and x = sum@pre and x > 0 and sum = sum@pre-x`
		The account is unlocked, it is neither full nor empty and a withdrawal is done that empties the account.

8.4 unlocked to unlocked	`sum@pre = 100 and sum = 0` `and x = sum@pre and x > 0 and sum = sum@pre-x`	The account in unlocked and full and a withdrawal is done that empties the account

Table 1: Test transitions for the method **withdraw (two transitions)**

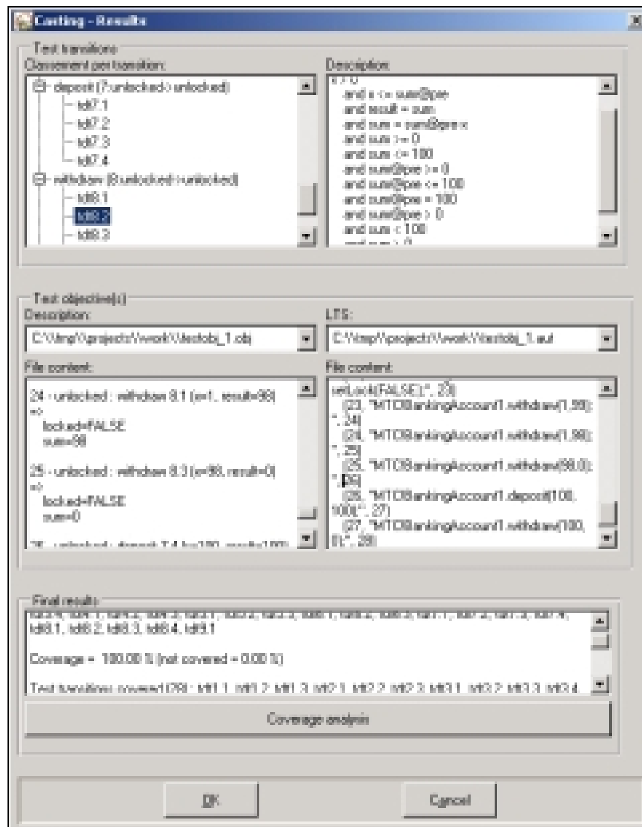

Figure 10: Presentation of the test generation results

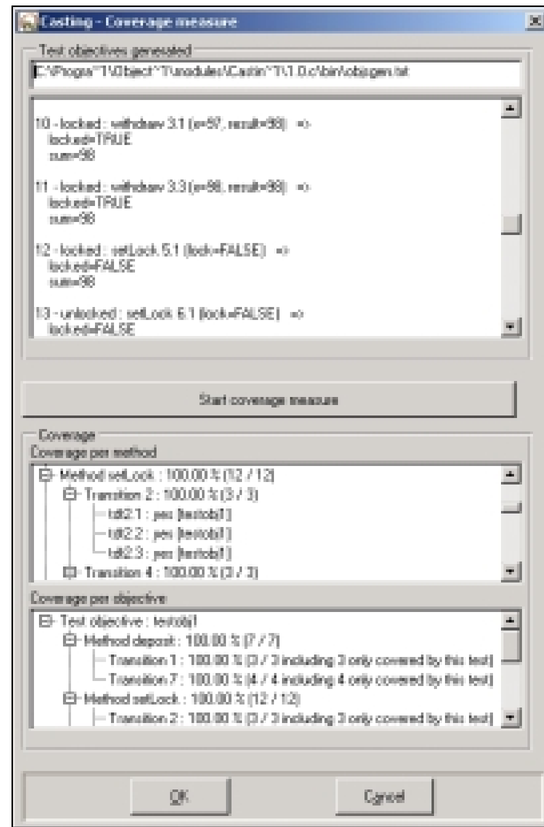

Figure 11: Coverage measure presentation

The test cases

The next step consists in asking the tool to search for test data actually satisfying the test transitions and used in valid test sequences, i.e. conforming to the state diagram control flow. These sequences also have to start from the initial state in order to facilitate their future integration and conversion into concrete (and executable) tests.

The tool, with control parameters (coverage rate = 100% and exploration depth = 30) has produced one test sequence that contains an instantiation for each test transition. This sequence is given in Table 2. By choosing a lesser value for the depth of search it is possible to obtain several test sequences instead of a long one.

Check performed in this sequence on the state resulting from the method calls is limited to return parameters of method calls. In fact, the objective here is not to build a concrete test suite including several checks but to identify test data satisfying a given set of constraint and leading the system to the appropriate input state. A solution to include more check in the test sequence is to define *get* methods for public attributes (getSum() and getLock() for instance), and by modifying the state diagram to include them after each method call in the control flow. These check methods will then be present in the test sequences and the conversion into a concrete test will be easier.

Coverage measures

Finally, the tool provides a coverage measure achieved by the produced test sequences. This measure is presented either for each method or for each test sequence. This is useful to determine the coverage achieved by a test sequence independently from the others. Moreover, the measure indicates if a test transition is covered only by the test sequence or if it is also covered by another test sequence.

n.	Input state	Method call and test transition	Test Transition	Expected account state
0	init	createAccount()	9.1	locked=FALSE, sum=0
1	unlocked	setLock(TRUE)	4.3	locked=TRUE, sum=0
2	locked	deposit(500,0)	1.3	locked=TRUE, sum=0
3	locked	setLock(TRUE)	2.3	locked=TRUE, sum=0
4	locked	setLock(FALSE)	5.3	locked=FALSE, sum=0
5	unlocked	setLock(FALSE)	6.3	locked=FALSE, sum=0
6	unlocked	deposit(98,98)	7.2	locked=FALSE, sum=98
7	unlocked	setLock(TRUE)	4.1	locked=TRUE, sum=98
8	locked	deposit(500,98)	1.1	locked=TRUE, sum=98
9	locked	setLock(TRUE)	2.1	locked=TRUE, sum=98
10	locked	withdraw(97,98)	3.1	locked=TRUE, sum=98
11	locked	withdraw(98,98)	3.3	locked=TRUE, sum=98
12	locked	setLock(FALSE)	5.1	locked=FALSE, sum=98
13	unlocked	setLock(FALSE)	6.1	locked=FALSE, sum=98
14	unlocked	deposit(1,99)	7.1	locked=FALSE, sum=99
15	unlocked	deposit(1,100)	7.3	locked=FALSE, sum=100
16	unlocked	setLock(TRUE)	4.2	locked=TRUE, sum=100
17	locked	deposit(500,100)	1.2	locked=TRUE, sum=100
18	locked	setLock(TRUE)	2.2	locked=TRUE, sum=100
19	locked	withdraw(99,100)	3.2	locked=TRUE, sum=100
20	locked	withdraw(100,100)	3.4	locked=TRUE, sum=100
21	locked	setLock(FALSE)	5.2	locked=FALSE, sum=100
22	unlocked	setLock(FALSE)	6.2	locked=FALSE, sum=100
23	unlocked	withdraw(1,99)	8.2	locked=FALSE, sum=99
24	unlocked	withdraw(1,98)	8.1	locked=FALSE, sum=98
25	unlocked	withdraw(98,0)	8.3	locked=FALSE, sum=0
26	unlocked	deposit(100,100)	7.4	locked=FALSE, sum=100
27	unlocked	withdraw(100,0)	8.4	locked=FALSE, sum=0

Table 2: Test sequence for the example

4. Related work

Test synthesis from UML models

There is a large body of work concerned with test synthesis form UML models. Here, we focus on those approaches that are concerned with the class and state diagrams. Offut and Abdurazik [19] define coverage criteria for testing class and state diagrams ("Logical Views" in their terminology) and detail a test generation tool, UMLTest that relies on these criteria. In particular, their "Full Predicate Coverage" criteria stipulates that tests must be generated for every (Boolean) clause c in a predicate P such that a change in the value of c can be observed as a change in the value of P. This criterion is stricter than ours, which just proposes to test each clause with the values true and false, irrespectively of the impact on the global value of the predicate. The language considered in [19] is restricted to only Boolean variables, which makes it possible to tabulate the entire state space, and reduces the search for test input (text prefixes) to a graph-reachability problem. Constraint resolution techniques like the one presented here would be needed to extend their approach to other types of data.

Kim *et al.* [16] show how to transform a UML state diagram into an extended finite state machine that makes explicit the control flow of a model and from which use-def data flow information can be extracted. Standard coverage criteria for control such as all-definitions or all def-use-paths can then be applied for identifying and generating test cases. The paper does not deal with the problem of finding test input to realize these test cases and no tool support is discussed.

The problem of producing test data (together with their expected outcome) has been addressed by Chevalley and Thévenod-Fosse [7] who use probabilistic methods to generate test data from UML state diagrams. Based on extensive simulation of the UML model, an input distribution is determined that describes which input will trigger a given transition. This, together with a test criterion based on transition coverage, allows to determine sequences of test cases for exercising the transitions of the diagrams under test. Compared to our work (and that of Kim *et al*. [16]) the simulation approach does not require any transformation of the UML model under test but its quality depends on being able to determine a "good" distribution. It is unclear to what extent this part of the process can be automated. The alternative (more in line with our approach) would be to

perform a static analysis of the diagram to determine an input distribution analytically but the authors discard this option as being unfeasible.

Jéron *et al.* [15] show how the simulation facilities of the UML tool UMLAUT can be used to obtain a transition system for which test cases can be generated by combining it with the test generation tool TGV [11]. Current work on the test case generation from UML models pursues this idea in the context of the French national project COTE [9] (as part of which the work reported here was accomplished) and the European IST project AGEDIS [4] concerned with test case generation from UML specifications by translating them into the intermediate language IF and apply the test generation tool TGV to this intermediate model.

Test data generation using Constraint Solving techniques

Constraint solving techniques for test case generation have been applied both to specifications and to programs. The seminal work of Bernot et al. [6] employs constraints to extract test cases from algebraic specifications. Dick and Faivre [10] propose a method for extracting tests from VDM specifications. Their approach is also based on transforming predicates occurring in the specification into disjunctive normal form from which a domain decomposition is deduced. This decomposition is then used to construct a finite state machine that serves to construct test cases in a semi-automatic fashion. In addition to the work by Van Aertryck *et al* ([1], [2] and [3]) on which the present work is based, several works have considered extracting test cases from B and Z specifications ([17], [20]). Common to these works is that they do not consider the decomposition of transitions in order to identify interesting test cases.

Gotlieb *et al*. ([12] and [13]) use constraints to model the behavior of a program by translating it into a constraint representation of its data and control flow. This system of constraints is then combined with constraints encoding criteria for structural testing of programs and fed to a constraint solver in order to determine the relevant test input. This work has been continued in the French national project INKA [14] that among other issues consider how to deal with floating point calculations. The results of this project would be highly relevant for extending our work to deal with floating point numbers which for the moment are not taken into account.

5. Implementation

We have developed a prototype implementing the UML-CASTING test generation method. The core of the tool (model parsing, attributes computation, search algorithm, constraint resolution) is written in SICStus Prolog. This implementation of Prolog provides libraries for constraint solving (clp(fd)) on finite domains. The tool has been integrated within Objecteering (a UML Case tool) as a UML Profile, taking advantage of facilities offered by Objecteering UML Profile Builder to connect an external tool to their UML Case tool.

The performance of the tool in terms of execution time is satisfactory. Computation time taken for constraint resolution for the complete test generation process (including test transition identification and test sequences generation) is less than two seconds for the example presented in this article.

6. Conclusion and future works

In this article we have presented the results of a first step toward an instantiation of the general test cases generation method CASTING to UML. Using this method it is possible to automatically produce test suites according to a given test strategy associated to a given coverage goal. This work has confirmed the generic nature of the CASTING test generation method through its applicability to another formalism than B, subject of its first instantiation. The test generation method presented offers a systematic approach and an automatic equivalence class identification, with an automatic test data synthesis and a test coverage measure. The actual adaptation of CASTING to UML can only manage a simple model and a subset of OCL. More work needs to be done in order to make this tool really useful to a test designer working with UML models. By restricting the kind of model that the tool takes as input, we have been able to study the whole chain of the test generation, starting from a class diagram and a state diagram and going to valued sequence of methods call.

As a test generation method will preferably not been used alone but in assistance and cooperation with classical and manual test design techniques, it is interesting to let the user define his own test cases and confront them with the intermediate representation obtained with CASTING. By doing this, we allow a complementary approach between manual test cases and automatically produced test cases. Test coverage measure of a test suite (a set of test sequences) is also a way to compare effective coverage achieved by several test suites that may have been produced accordingly to different testing techniques.

We have identified several source of improvement for the test generation process. Already defined test strategies, even if simple ones, are sufficient to generate interesting tests cases but new and more specific test strategies should be defined to cover other testing practices like boundary testing. Data structure related test can be obtained by providing a test strategy implementing the regularity hypothesis (defined in [6]) which is another formal representation of a test hypothesis related to data structures. A test strategy that takes into account the order of method calls would certainly be a great improvement. Concerning the boundary testing, a user can use the tool without any modification to produce boundary test. To do so, he would exploit the possibility of OCL and systematically add constraint on domains range for all the variables. This modification to the model, in conjunction with an appropriate test strategy will generate boundary tests. Defining test objective in a more fine-grained manner than just a coverage percentage should also be a useful improvement. This could be done using a finite state automaton (refusal states, etc...), in order to provide a guide to the search for testing path. Use cases might also be a way to express these test objectives. It is not clear for now if the user will be interested in defining his test strategy at the level of the decomposition rules. Predefined test strategy with little control may be sufficient but this has to be thought about. In fact test strategies are only one part of the technique to generate test cases. The way the intermediate representation is used to generate the test sequences (covering all the test transitions, starting from the initial state) does have a great impact on the kind of generated tests. Allowing the user to control this step of the test generation process, by, for example defining sub-sequences that he wants to be covered, is another way to improve the method. The integration of user made test sequences with the generation of the test sequences that complete them to achieve a given coverage measure should also be useful in a mixed manual/automatic approach to test generation.

Another domain of improvement is to extend the UML model taken as input. First, dealing with multi classes' models and more data structure (like array) appear to be the most urgent tasks. To do this, we have to extend the input constraint language by dealing with arrays, sets, collections and floats. This should be done conjointly with the definition of new elementary testing techniques for these operators. The use of object diagrams to define an initial system state instead of having to compute one using methods and constraint resolution may be a way to spend more computation time on productive aspect of test synthesis. In a "full UML" perspective, the use of Message Sequence Chart (MSC) to present the produced test sequences to the user seems to be widely accepted.

There is a lot of domains in information technologies for which test generation is of interest. We are interested in studying what our method can add to a certification process as it is done within the Common Criteria [8]. More precisely, we plan to study the adaptation of our method to specificities of Common Criteria testing requirements in order to produce automatically expected and adequate evidences for components of the assurance class ATE. That assurance class cover all the testing aspects of the development of a product. Also for the Common Criteria the possibility to generate automatically valid sequences might help on the correspondence relation between two levels of specification. Indeed, if it is possible to generate equivalent sequences from two levels of specification, this may be consider as evidence that both representations of the model are equivalent (without of course the intent to prove it that way). Test cases generated from the model can be viewed as use cases at a specification level, showing a possible behavior of the model and thus of the implementation that it is supposed to be a correct refinement of.

References

[1] L. Van Aertryck.. "Une méthode et un outil pour l'aide à la génération de jeux de tests de logiciels". PhD thesis, Université de Rennes I, 1998. (ftp://ftp.irisa.fr/techreports/theses/1998/van_aertryck.ps.gz)

[2] L. Van Aertryck, M. Benveniste et D. Le Métayer. "CASTING : une méthode formelle pour la génération de tests". Proc. AFADL (Approches Formelles dans l'Assistance au Développement de Logiciel), pp 99-112, 1997, Toulouse, France.

[3] L. Van Aertryck, M. Benveniste et D. Le Métayer. *CASTING: A formally based software test generation method*. IEEE International Conference on Formal Engineering Methods (ICFEM'97), 1997, Hiroshima, Japon.

[4] AGEDIS (Automated Generation and Execution of Test Suites for DIstributed Component-based Software), European IST project (www.agedis.de).

[5] B. Beizer. Software Testing Techniques, Van Nostrand Reinhold, New York, 2nd ed., 1990.

[6] G. Bernot, M-C. Gaudel and B. Marre, Software testing based on formal specifications: a theory and a tool, Software Engineering Journal 6:387-405, 1991.

[7] P. Chevalley and P. Thévenod-Fosse, Automated Generation of Statistical Test Cases from UML State Diagrams, Proc. Of the 25th Int. Computer Software and Applications Conference (COMPSAC'01), pp. 205-214, IEEE Press, 2001.

[8] Common Criteria for Information Technology Security Evaluation (http://www.commoncriteria.org/).

[9] COTE (Test de composants), project funded by the French "Reseau national de technologies logicielles", 2000-2002. (www.irisa.fr/cote).

[10] J. Dick and A. Faivre, Automating the generation and sequencing of test case from model-based specifications, Proc. of Industrial Strengh Formal Methods (FME'93), (eds. Woodcock, J., Larsen, P.), Springer LNCS vol. 670, pp. 268-284, 1993.
[11] J-C. Fernandez, C. Jard, T. Jeron, C. Viho, Using On-the-fly Techniques for the Generation of Test Suites, CAV'96, 1996.
[12] A. Gotlieb, Génération automatique de cas de test structurel avec la programmation logique par contraintes, PhD thesis, Université de Nice-Sophia Antipolis, 2000.
[13] A. Gotlieb, B. Botella, M. Rueher, Automatic Test Data Generation using Constraint Solving Techniques, Proc. ISSTA 98 (Symposium on Software Testing and Analysis, ACM SIGSOFT, vol. 2, pp. 53-62, 1998.
[14] INKA (Génération automatique déterministe de données de test selon des critères de couverture structurelle) Project funded by the French "Reseau national de technologies logicielles", 2000-2002.
[15] T. Jéron, J-M. Jezequel, A. Le Guennec, Validation and Test Generation for Object-Oriented Distributed Software. PDSE'98, pp. 51-60, 1998 IEEE.
[16] Y.G. Kim, H.S. Hong, S.M. Cho, D.H. Bae, S.D. Cha, Test Cases Generation from UML State Diagrams, IEEE Proceedings 146(4), pp. 187-192, 1999.
[17] B. Legeard, F. Peureux, and M. Utting, Generation of functional test sequences from B formal specifications - Presentation and industrial case study. In 16th IEEE International Conference on Automated Software Engineering (ASE'01), San Diego, USA, November 2001.
[18] OCL UML v1.4 Draft, February 2001, Chapter 6: Object Constraint Language Specification.
[19] J. Offutt and A. Abdurazik, Generating Tests from UML Specifications, Proc. of UML'99, Springer LNCS vol. 1723, pp. 416-429, 1999.
[20] A. Pretschner. Classical search strategies for test case generation with Constraint Logic Programming. In BRICS, editor, *Proceedings of the Workshop on Formal Approaches to Testing of Software (FATES'01)*, pp. 47-60, Aalborg, Denmark, 2001.
[21] Object Management Group. Unified Modeling Language (UML 1.4, September 2001. http://www.omg.org/cgi-bin/doc?formal/01-09-67.