SOFTWARE TESTING, VERIFICATION AND RELIABILITY
Softw. Test. Verif. Reliab. 2014; **24**:499–531
Published online 29 May 2014 in Wiley Online Library (wileyonlinelibrary.com). DOI: 10.1002/stvr.1537

Test generation with Satisfiability Modulo Theories solvers in model-based testing

Jérôme Cantenot, Fabrice Ambert*,† and Fabrice Bouquet

FEMTO-ST dpt DISC, Université de franche-Comté, Besançon, France

SUMMARY

This paper presents a framework for UML/OCL model test generation. This framework is based on three parts. The first part is the transformation from a UML/OCL model into a Satisfiability Modulo Theories (SMT) instance. The metamodel associated with the two models and the transformation rules are presented. The second part is the animation strategies used for the generation of SMT instances to compute test sequences. The paper studies five different types of strategy to build the first-order formula used by an SMT instance. The paper also gives the results of experiments on six case studies that were used to validate the method. Copyright © 2014 John Wiley & Sons, Ltd.

Received 16 December 2012; Revised 24 April 2014; Accepted 25 April 2014

KEY WORDS: SMT solver; model-based testing; UML/OCL

1. INTRODUCTION

During the last decade, test activity has become a major challenge for many companies. The predominant use of UML as a modelling tool has led many players to focus on the production of tests from UML models. Model-based testing (MBT) from UML is an answer to this problem.

In this context, the following approach aims at generating automatically test cases to stimulate each of the operations and transitions of the model. For this, only the following three UML diagrams are considered: class diagram, object diagram and state diagram. The class diagram describes on the one hand the static model. On the other hand, operations of classes describe the dynamics of the system. The object diagram instantiates the initial state of the system. The state diagram completes the description of the system dynamics. On the basis of a model limited to these three diagrams, the proposed method is based on the use of a Satisfiability Modulo Theories (SMT) solver. For this, a translation of the model elements into a set of first-order formulas capturing static and dynamic elements of the model is proposed. Resolution by the SMT prover allows the building of operations and transition activation suites to cover the model elements.

The first step of the generation process is the translation of the original model into first-order formulas. The first contribution concerns the definition of a metamodel SMT and transformation rules from UML4MBT metamodel to SMT metamodel. The UML4MBT metamodel is dedicated to MBT and brings restrictions from a classical UML metamodel [1].

An instance of an SMT model describes a succession of states of the system in a number of steps. It is supplemented by a test target to reach, which may correspond to an operation to activate, a transition to pass or a state to reach. From these elements, the SMT solver attempts to produce a solution. This solution can be used to build some test cases.

*Correspondence to: Frabrice Ambert, FEMTO-ST dpt DISC, Université de franche-Comté, Besançon, France.
†E-mail: fabrice.ambert@femto-st.fr

Considering each test target independently, the test generation process needs to solve several times the same formulas. Therefore, in order to obtain the whole coverage of the model, experiments with several strategies to generate tests are performed. One approach is to chain several SMT instances of reduced size where the starting state of an animation is the final state of a previous animation. To improve the efficiency of the strategies, a framework to parallelize the SMT resolution processes is proposed. To study the performance of the method, experiments are conducted on several models of different types.

The rest of the paper is organized as follows. The first part presents the SMT context of the work, the chosen solvers and the chosen theory. The second part deals with the presentation of the modelling language, that is UML4MBT/OCL4MBT. The next section describes the running example. In the fourth section, the test generation process is exposed. This is the support for the introduction of the fifth part, the SMT metamodel definition. The sixth part is devoted to a presentation of the translation rules from the UML4MBT metamodel to the SMT metamodel. Section 8 reports the different strategies, and the experiments take place in the next section. To finish, a comparison between the presented approach and some works is presented before concluding.

2. SATISFIABILITY MODULO THEORIES

An SMT problem aims at determining if a set of logical formulas, named an SMT instance, is satisfied. The formulas are expressed in first-order logic (FOL), where some function and predicate symbols have specific interpretations defined in a background theory. Indeed, the theories in SMT stand for the use of a combination of background theories, such as the theory of integers, the theory of real numbers or even the theory of bit vectors, in the same formula. These multiple theories make it possible to have good expressivity.

The solvers used during the method to generate tests automatically from a model are presented first. Then the common input language for the SMT solvers presentation follows. Finally, the chosen logic is detailed.

2.1. *Satisfiability Modulo Theories solvers*

In the recent years, the efficiency of SMT solvers has greatly improved. This progress is monitored through an annual competition, named SMT-COMP [2]. This competition is possible because a common language for SMT provers, named SMT-LIB [3], has been created. This language allows one to define a background theory and an SMT instance.

For the experiments, the chosen solvers are Z3 [4] and CVC3 [5] because they have good performances in the QF_UFLRA [6] logic that allows the use of closed quantifier-free and linear formulas over the theory of integer arrays with free sort, function and predicate symbols. Moreover, these solvers support all the major theories defined in the SMT-LIB language.

2.2. *SMT-LIB*

The SMT-LIB language offers the possibility to represent a problem in first-order logic through an SMT instance comprehensible by a large set of SMT solvers. The presented method uses version 1.2 of the language. An SMT instance contains the following elements:

Logic The logic used to solve the SMT instance must be specified. In our case, the QF_UFLRA logic is used, detailed in the next subsection.

Status If the existence or the absence of solution to the SMT instance is known, the information of status can increase the solver efficiency.

Sort These elements allow the definitions of new types.

Function The functions can be variables or mathematical operators.

Assumption The assumptions are a list of constraints that are considered true.

Formula There is only one formula in an SMT instance. The formula is a predicate that the SMT solver tries to prove or disprove.

Solving the SMT instance by the solver has three possible outcomes. The first outcome is that the solver has not been capable to determine if a solution exists. In this case, a reformulation of the problem, especially of the formula containing quantifiers, can sometimes resolve this problem. The second is that there is no solution. The third occurs when a solver finds a solution. In this case, the solver can give a valuation of the functions respecting all assumptions and the formula.

2.3. *QF_UFLRA logic*

The chosen logic is the QF_UFLRA logic (unquantified linear real arithmetic with uninterpreted sort and function symbols). This logic has closed quantifier-free formulas built over arbitrary expansions of the real signatures with free sort and function symbols but containing only linear atoms, that is atoms with no occurrences of the function symbols * and /, except if terms have concrete coefficients. Moreover, this logic has the following extensions:

- The syntax $(+t_1 \dots t_n)$ with $n > 2$ is allowed as an abbreviation of the term $(+t_1\,(+t_2\,(+t_3 \dots)))$.
- Terms with positive integer coefficients are allowed, that is expressions of the form $(*nt)$ or $(*tn)$ for some numeral n, both of which abbreviate the term $(+t \dots t)$ having n occurrences of t.
- Terms with negative integer coefficients are also allowed, that is expressions of the form $(*(-n)t)$ or $(*t(-n))$ for some numeral n, both of which abbreviate the term $(-(*nt))$.
- In addition to 0 and 1, numerals $n > 1$ are allowed also as arguments of atoms or of $+$. In that case, they abbreviate the term $(+1 \dots 1)$ with n occurrences of 1.

The QF_UFLRA logic defines the types–integer, real and boolean–and the operators–not, $\Rightarrow$, and, or, xor, $=$, *distinct*, *ite* (if then else), $-$, $+$, $*$, div, mod, abs, $\leq$, $\geq$, $<$ and $>$.

The choice of this logic was made in regard to three criteria:

- An important criterion is the capacity of the solver to conclude. Indeed, by default, when a problem is submitted to a solver, there are three possible outcomes: sat (there is at least a solution), unsat (there is no solution) and unknown. The solver cannot always decide if there exists a solution when formulas are not quantifier free because there is no sound and complete procedure for FOL formulas of linear arithmetic with uninterpreted function symbols [7]. To remove this problem, a logic without quantifiers has been chosen.
- Another criterion is to prefer a logic efficiently implemented by a large number of solvers. An annual competition between the SMT solvers, named SMT-Comp, provides data to evaluate the performance of the state of the art solvers. This competition shows that this logic is implemented by the major solvers like Z3 [4], CVC3 [5] and YICES [8].
- The last criterion is the expressiveness of the logic. The logic must encode necessary information to represent the animation of the system under test (SUT) with a transition system. In particular, the logic must allow the creation of a mechanism to represent the set type and the basic set operators.

3. LANGUAGES FOR MODEL-BASED TESTING

To model a system, a language adapted to the model-based testing has been chosen. This language is defined by Bouquet *et al.* [9]. It is a subset of UML/OCL. The following section provides a reminder of this language for the UML part and the OCL part.

3.1. *UML4MBT*

To model a system, a language adapted to model-based testing (MBT) has been chosen. This language, named UML4MBT for UML for model-based testing, is defined as a subset of UML 2.0 by Bouquet *et al.* [9]. To fulfil the needs of test generation, this language has taken a specific point of view that is articulated around the notion of SUT.

The selection of the UML4MBT language is due to the following reasons:

- Test generation is a core concept of the language.
- The language is derived from a standard: UML.
- A system can be modelled with standard industrial tools like Topcased or Rational Software Architect.
- A previous model can be converted more easily with the support of the conversion between XMI (XML Metada Interchange) and UML4MBT.
- The language supports the Eclipse Modelling Framework [10].
- There are previous examples of projects modelling complex and industrial systems. For example, a smart card module in the European SecureChange project [11] has been modelled.

The specific point of view on the system and the test generation goal result in the limitation of the authorized diagrams. From the 13 diagrams included in UML 2.0, only three are allowed. The first one is the class diagram. This diagram is mandatory and models the system with a list of classes, attributes, operations and associations. Within a model-based testing process, there is a tendency to model around a specific class representing the SUT. However, this approach is not mandatory. By definition, this diagram gives information about the structure, or static part, of the system. But it can also describe the behaviour of the system through the operations of the classes.

Another solution to model the dynamic part of the system is to use a different diagram: the state-chart. This diagram is optional because the behaviour of the system can be entirely encoded through the pre/post condition found in the operations defined in the class diagram. However, the behaviour is usually described in both diagrams. An advantage of the state-chart is to facilitate the representation of the final states of the system and that it supports a more natural way to represent a sequence of states of the system. Indeed, in order to represent a final state using a class diagram, all preconditions of the operations have to be modified.

The last diagram is the object diagram, which is used to represent a state of the system with the instances and links between entities. A partial state of the system can be defined if some slots (instances of attributes) and links (instances of associations) are not valued.

The UML4MBT language has the following restrictions in regard to UML 2.0:

- The inheritance relationship is forbidden. As a consequence, polymorphism is also removed.
- The real and string types are forbidden. A solution to represent string values is to use enumeration.
- The integer variables have a range. If *n* range is not specified, a default value is applied.
- The sets contain only instances. If a set of integers is necessary, then a class modelling an integer must be created.
- All instances used by the system during the execution of any behaviour must be defined in the object diagram. Consequently, an object diagram has two goals. The first is to describe a view of the system at an instant *t*. The second is to detail all object instances used during the execution.

These restrictions are acceptable to model real life systems that are finite and have the advantage of defining a system with a limited number of states that can always be enumerated.

3.2. *OCL4MBT*

To model the dynamic part of the system, a constraint language, named OCL4MBT for OCL for model-based testing, is used in conjunction with the UML4MBT language. Therefore, OCL4MBT is used in the pre/post of the operations defined in the class diagram and in the guards and actions of the transitions defined in the state-chart.

This language is derived from OCL but is not a subset because the semantics of the OCL operators is not preserved in all cases. This language has been created specifically for model-based testing and must be understandable by a common tester. Hence, OCL4MBT does not respect the standard preparadigm/postparadigm but uses two contexts: Assignment and Evaluation.

The first context, named Evaluation, respects the standard semantics of the OCL language and can be found in the preconditions of the operations, the guards of the transitions and the conditions in the 'if then else' operators. Therefore, in this context, $a = b$ means 'is the value of a equal to the value of b?'.

The second context, named Assignment, transforms OCL into an imperative, left to right evaluated language. This context can be found in the postconditions of the operations and the actions of the transitions. Therefore, in this context, $a = b$ means 'a takes the value of b'. This context has been created to facilitate the work of the tester who can write $a = 1\ and\ a = a + 1$ and obtain a predicate evaluated to true and equivalent to $a = 2$.

Some operators are not available in both these contexts. The assignment context allows only operators that do not offer a choice that makes the model nondeterministic. For example, the operator or is only available in the evaluation context, but the operator = is used in both contexts.

To conform to the restrictions imposed by the UML4MBT language, some operators have been modified from the standard OCL operators. In particular, the set operators can only be applied on collections containing instances. Moreover, the 'collect' operator can only return a collection of instances. This operator returns a collection of instances obtained by applying a formula on each instance of a collection.

4. RUNNING EXAMPLE: ROBOT

This robot is an automated arm that can transfer items between three conveyor belts. This system is constituted by four parts:

- The incoming conveyor belt, located in the bottom-left corner of the room, brings new items. There are three kinds of item, named T1, T2 and T3, and a conveyor belt can only contain one item.
- An outgoing conveyor belt, located in the upper-left corner of the room, evacuates T1 and T3 items and can only contain one item.
- An outgoing conveyor belt, located in the upper-right corner of the room, evacuates T2 and T3 items and can only contain one item.
- An arm transfers the items. The arm can rotate between left and right positions, translate between up and down positions, grab an item and drop an item.

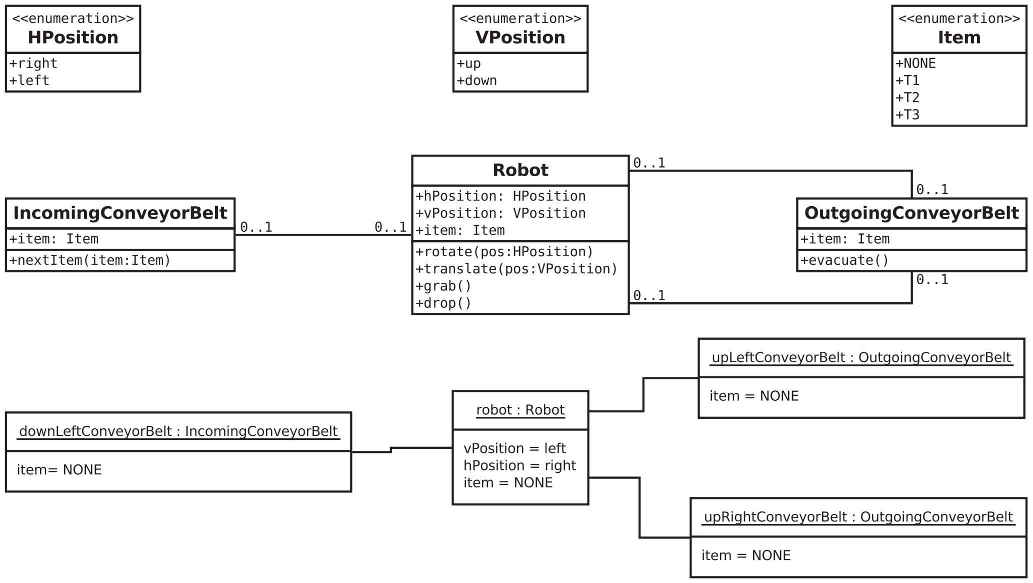


Figure 1. Class and object diagrams of the Robot model.

To increase the performance of the system, the arm must always minimize the number of moves to transfer an item. Therefore, the arm cannot rotate three times successively for example.

The model of the robot, written in the UML4MBT/OCL4MBT languages, uses the class and object diagrams. These diagrams are given in Figure 1. This model highlights the properties of these languages. For example, inheritance relationship is forbidden. That is why the different items are modelled by an enumeration. Moreover, this model can only take a finite number of states. All operations of the model are described in OCL4MBT and are given in Appendix A.

5. TEST GENERATION PROCESS

The core idea behind the automatic test generation process with an SMT solver is to represent the behaviours of the system and the test targets in first-order logic formulas compatible with the solver. The process has four main phases: encoding the animation of the system and the test targets into an adapted model, translating this model into a language compatible with an SMT solver, solving of the problem and building tests from the solution found by the solver.

The first part will present the global process used to generate tests automatically from a model written in UML4MBT, a subset of UML. Second, the details of how to encode the animation of the system within a transition system are given. Third, the notion of test case and test target are defined in regards to the transition system.

5.1. Global process

Figure 2 shows the process to generate tests automatically from a UML4MBT file that models a system. This process uses the following modules:

1. The *UML4MBT transformer* modifies and adds information to the model. The transformer adds metainformation. In particular, the elementary behaviours are created. These behaviours are described in Section 5.3. To activate a behaviour, a precondition represented by a boolean must be activated.
2. The *SMT Converter* creates an SMT instance from a UML4MBT model and a list of parameters: the number of steps (also known as the length of the animation), the targets to reach and the initial state of the model. The metamodel defining an SMT instance is described in Section 6. To preserve a link between the UML4MBT and the SMT entities, a dictionary is created.
3. The *SMT writer* writes the SMT instance in a file. This file respects the SMT-LIB v1.2 syntax. This module is also responsible for adapting the syntax when a solver does not conform to the standard.

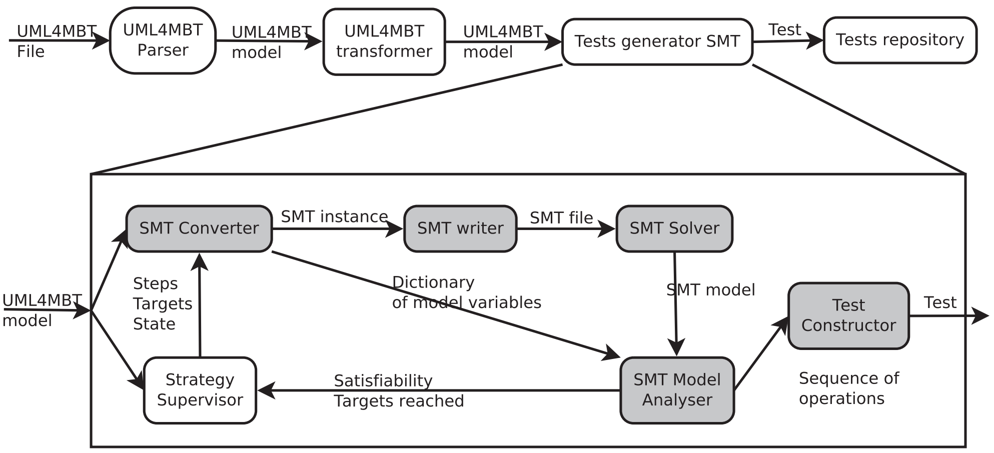


Figure 2. Workflow of the test generation process with a Satisfiability Modulo Theories solver.

4. The *SMT Solver* solves the SMT instance if possible. The SMT-LIB standard gives a level of abstraction that allows any SMT solver to be employed without modification. However, a solver must support the QF-UFLRA logic to be usable in the presented process.
5. The *SMT Model Analyser* analyses the solutions to extract the following information: the satisfiability of the SMT instance, the list of the executed operations and their parameters and the list of targets found. To obtain this information, a dictionary associating the variables of the model to their SMT function counterparts is necessary. This dictionary is created during the conversion by the SMT Converter.
6. The *Strategy supervisor* selects which SMT instance must be created. In particular, the supervisor chooses the list of targets to reach and the length of the animation. The supervisor also decides when the test generation is finished because a time limit is met or all test targets are found.
7. The *Test constructor* creates the test cases from the valid transition sequences found by the SMT solver.

This architecture can simultaneously solve different SMT instances with an SMT solver. In Figure 2, the gray box indicates which module can be parallelized. In this configuration, the strategy supervisor also manages the life cycle of the threads.

5.2. Animation

The SMT-LIB language does not have the mechanisms to add specific information to the different logic formulas used to describe the problem. To solve this difficulty, a metamodel is created to adapt the SMT-LIB language to test generation through the model animation. This metamodel adds semantics for the tests to the SMT-LIB formulas and variables. The metamodel is an intermediary phase between the semantics present in the UML4MBT/OCL4MBT languages and the syntax used by the SMT-LIB language. The metamodel will be described in details in Section 6.

The metamodel depends on the animation process of the SUT. The animation is encoded through a basic transition system derived from the UML4MBT model. A notation derived from the one described by Fraser *et al.* [12] is adopted. In this framework, a state of the system is determined by the valuation of n variables $\{x_1, x_2, \ldots, x_n\}$, named state variables. For each variable x_i, there is a corresponding domain D_i. Consequently, the search space of the system is defined by $D = D_1 \times D_2 \times \ldots \times D_n$.

The valuation of a state is defined by a logic formula ρ. This state can be the initial state of the system described by the object diagram of the UML4MBT model or a state found during a previous valid animation.

The valid transitions between two states are represented by a transition relationship that is a subset of $D \times D$. A transition can be derived from an operation of the class diagram or a transition of the state-chart. A transition δ is a relation between two states. Δ is the set of all transitions δ. In this paper, to differentiate a transition from the UML4MBT state-chart and a transition of the transition system, a transition from the state-chart will be called a UML4MBT transition. In a state, the state variables are divided into two categories: X_a regroups the variables found in the action of the transition and X_b contains the remaining state variables. To illustrate this process, the operation *nextItem* of the Robot model is considered. This operation adds a new item of the selected type to the incoming conveyor belt if this conveyor belt does not have an item. In this case, X_a contains only one variable that corresponds to the slot *item* of the instance *downLeftConveyorBelt* and X_b regroups the variables encoding all other slots and links. The transition δ is defined by the relation $\delta = \alpha \land \beta \land \gamma$ where α sets the values for the variables X_a, β defines the new values of the state variables X_b after the execution of the transition and γ is the guard of the transition. In the operation *nextItem*, α sets the value of *downLeftConveyorBelt.item* to the value of the parameter *item*, β forces the other states variables to keep their previous value and γ tests if *downLeftConveyorBelt.item* is equal to *Item.NONE*. Moreover, an additional constraint is necessary to suppress the possibility of executing two transitions simultaneously. This constraint is added only as a precaution in case of errors during the creation of the model. The UML4MBT language only allows deterministic models, so that the generated test sequences can be controlled by their inputs.

5.3. Test

To write a test target about a specific behaviour, the transition must be decomposed into elementary behaviours. In his book, Beizer [13] describes the most common methods to rewrite the conditions of the transitions to explicit elementary behaviours. These methods lead to create a predicate $cpt_{behaviour}$ representing the condition to activate the elementary *behaviour*. Considering the operation $rotate$ of the running example, two elementary behaviours are obtained. Indeed, the behaviour of the arm differs if it turns to the left or the right. Consequently, two predicates are created: $cpt_{rotate-left}$ and $cpt_{rotate-right}$. On the contrary, the operation $grab$ defines only one behaviour and so only one predicate: cpt_{grab}.

Therefore, a test target is a logical formula constructed from the predicates $cpt_{behaviour}$, the states of the system that can be partial and the transitions Δ. An example of test target, named ϕ_1, for the robot is 'op^*, *rotate*, *upLeftConveyorBelt.item* $= T1$' that can be read as 'execute any operation n times', and then the arm must execute a rotation to obtain a state of the system where the left outgoing conveyor belt contains a T1 item. Another kind of test targets is '$cpt_{rotate-right}$' that can be read as 'execute any transition to perform the behaviour corresponding to a rotation to the right'. This target is named ϕ_2. Therefore, this last example is equivalent to '$op^*, cpt_{rotate-right}, op^*$'.

The notion of test goal is also defined. The test goal indicates which test target must be reached during a specific animation of the system. This notion is necessary because the following constraint cannot be expressed in first-order logic: *during an animation, the maximum number of test targets must be reached.* Indeed, it is not possible to simply give all test targets to the solver and ask it to find the corresponding test cases. For example, some behaviours can be mutually exclusive. Therefore, the test goal is generally defined with one of these constraints: 'reach all targets of a list', 'reach a specific target' and 'reach at least one target from a list of targets'. Considering the two test targets given in the previous paragraph, the three kinds of test goal correspond, respectively, to $\phi_1 \wedge \phi_2$, ϕ_1 (or ϕ_2) and $\phi_1 \vee \phi_2$.

With the notation adopted in the previous subsection, the following definitions can be established:

- A test case $t = \langle s_0, s_1, \dots, s_l \rangle$ for a transition system $M = (D, \Delta, \rho)$ is a finite sequence with $0 \leq i < l : \Delta \models (s_i, s_{i+1})$ and $s_0 \in D$.
- A test target is a predicate noted ϕ. The set of the test targets is noted by Ψ.
- A subpath t_i of test case t is a subpath starting from the state s_i.
- If there exists a subpath t_i of test case $t = \langle s_0, s_1, \dots, s_l \rangle$ respecting $t_i \models \phi$ with $0 \leq i \leq l$ then the test target ϕ is satisfied, else the target is unreachable with animations of the length l.

6. METAMODEL

The core idea of the general process for test generation with the SMT solvers is to encode both the animation of the SUT and the corresponding test cases in first-order logic formulas. However, the language used by the SMT solvers is a very low level language and does not support the necessary concepts to represent the tests and the animation. Therefore, there is a need to create a new metamodel that has the capacity to represent the animation of the system with a transition system.

A first version of the SMT metamodel has been proposed in the paper [14]. The second version described here adds metainformation to the different entities to store their goals. With this information, it is now possible to select only some constraints in a model in function of the animation sequence to be executed. This part will detail the guidelines used during the creation of the SMT metamodel and the resulting architecture.

6.1. Guidelines

The metamodel must be able to encode three kinds of information. First, the metamodel must encode the necessary information to animate the SUT. It is represented by a transition system $M = (D, \Delta, \rho)$ where D is the search space, Δ is the set of the transitions δ and ρ is a formula to initialize the system. Second, the metamodel must encode the test targets Ψ and one test goal. Third, the metamodel must have the capability to extract the different test cases from the solution found by the solver.

The architecture of the metamodel is governed by three guidelines. The first is to stay as close as possible to the syntax proposed by the QF_UFLRA logic. Indeed, if a maximum of entities are derived from this syntax, the conversion of a model into a set of formulas using the SMT-LIB language will be greatly simplified.

The second guideline is to add information to the variables and formulas through metadata. By definition, the metadata concerns pointless information to solve the problem with an SMT solver but useful data to generate test cases. For example, a boolean variable b_1 is defined in the model and this variable is usable by the solver without any supplementary information. However, if this variable is also a test target, this information is preserved through metadata.

The third and last guideline is to create only the entities necessary to represent the animation of the system described in an UML4MBT language. Indeed, this language has some important restrictions with respect to UML that have direct consequences on the metamodel. For example, the UML4MBT does not allow the real type or the inheritance relationship.

6.2. Architecture

The architecture is shown on the class diagram (see Figure 3) and is articulated around four major kinds of entities: the types, the functions, the constraints and the instance.

Four different types are defined in the metamodel: *Integer*, *Boolean*, *Named* and *Set*. The *named* types are defined by the user and support only the equality operator. This kind of type is used to represent a class or an enumeration found in the class diagram of the UML4MBT model. This information is necessary to create the test case from the solutions given by the solver and is stored in a metadata. The *set* types need to contain only elements of the named type due to the restrictions imposed by UML4MBT.

The functions of the metamodel are classified under three categories: *constant*, *variable* and *stepVariable*. A type is associated to each function.

The *constants* can be an integer number, the ‘true’ value, the ‘false’ value, a set of named elements representing all instances of a class and a named value. In the case of named values, the name of the corresponding UML4MBT element is stored with a metadata. The named constant is only present for semantic reasons. Indeed, from the point of view of the solver, a named constant and a named variable are indistinguishable. For example, the constant ‘Position’ representing a slot of the instance ‘robot’ and the constant ‘left’ representing a literal of enumeration ‘HPosition’ share a common representation in the SMT-LIB language. However, from the point of view of the system, these functions have different roles: ‘Position’ represents a variable encoding the horizontal position of the robot, and ‘left’ corresponds to a constant encoding one possible position. The introduction of an artificial distinction preserves the meaning of the different functions.

The *variables* can be of any type and are used to represent a function that takes only one value during the animation. For example, a variable can represent a test target.

The *StepVariables* differ from the variables because they can take a different value after the execution of a transition. Moreover, these functions are categorized between the state variables, the input variables, the animation variables and the test targets. The input variables are found in the condition of the transitions and are not useful to define the state of the system. The animation variables only concern the animation process. For example, a stepVariable can store the sequence of the activated transitions. The combination of the input and animation variables allows to determine the path associated to the test case.

A *SMT instance* is divided into three elements:

- The first one is the logic. By default, QF_UFLRA is used, but it is possible to select a more restrictive logic if it is more adequate to the system.
- The second one is a set of assumptions. An assumption is a logical formula that is considered to be true during the resolution process. The objective of the assumption is stored. The objective can be ‘static’, ‘test’, ‘transition’ or ‘animation’. The ‘static’ assumptions correspond to the static constraints issued from the class diagram. An example would be the definition of

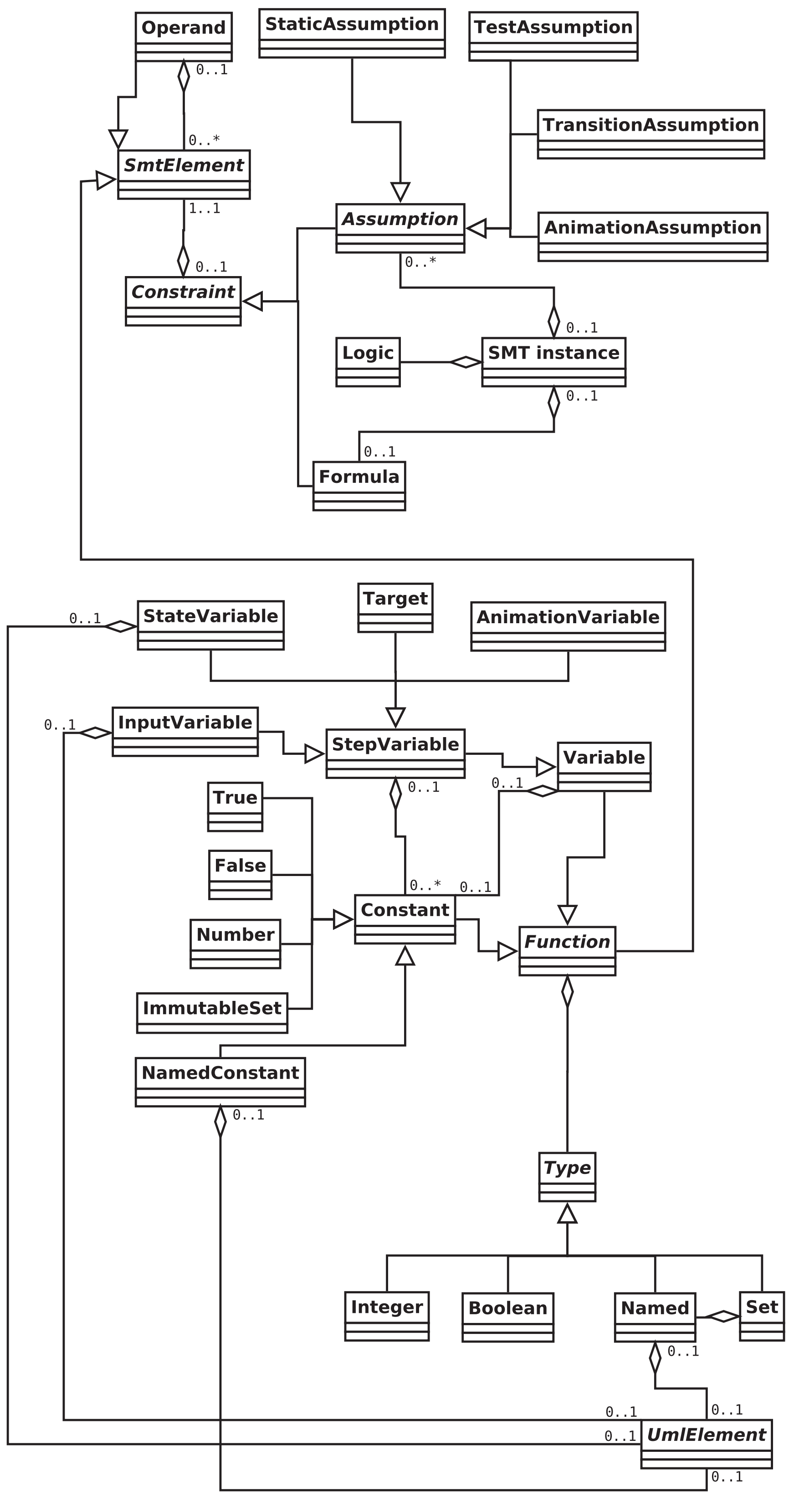


Figure 3. Metamodel of Satisfiability Modulo Theories for the test.

a domain for an integer attribute. The 'test' assumptions specify the condition to reach a test target. The 'transition' assumptions encode the transitions Δ of the transition system M. The 'animation' assumptions force the solver to execute a sequence of transitions.

- The third and last element is the Formula. In an SMT instance, the Formula is a constraint that can either be satisfiable or not under the assumptions. In our case, the Formula encodes the test goal. As a consequence, if two animations have the same length and employ the same transitions, then their SMT instance only differs by their formulae.

The metamodel must also be capable of representing the solutions of an SMT instance consisting of a valuation of the functions. The different values are encoded into constants. The stepVariables need a list of values of the same length as the number of executed transitions. These values and the stored metadata allow to obtain a sequence of the states of the system and the test cases.

In order to convert an SMT model into an SMT instance written in SMT-LIB, one needs to consider how to represent a collection and the set operators. Indeed the QF_UFLRA logic cannot natively express constraints on the set theory. The work of Kroening and Rü [15] adds this theory using the array theory and some axioms, but they use quantifiers. To overcome this problem, the set that contains current instances of class C is converted into a mathematical application $A : inst(C) \to \{0, 1\}$, where $inst(C)$ is the set of all instances of class C. Thus, the application A associates 1 if the instance is actually present in the set, and 0 otherwise. These applications can be understood as membership functions, and the corresponding operator is has.

Finally, the metamodel has the following operators: not, $\Rightarrow$, and, or, distinct, ite, $=$, $-$, $+$, $*$, div, mod, $\leq$, $\geq$, $<$, $>$ and has.

The metamodel is not strongly typed. This choice allows more factorized code but can lead to the creation of models without structural integrity. For example, it is technically possible to compare an integer function with a boolean function. To remove this problem, the instantiation of a model is delegated to a specific class that can check for structural errors.

7. CONVERSION RULES

To convert a system modelled with a UML4MBT model into an SMT model, a list of conversion rules is established. A first version of the rules has been proposed in paper [14]. The new version takes into account the modification of the SMT metamodel. First, the notation used to formalize the rules is introduced. Then the rules to convert the UML4MBT and the OCL4MBT elements will be presented successively. Finally, the rules governing the transition system are introduced.

7.1. Notation

To define the conversion rules, the UML4MBT model is considered as an eight-tuple (Class, Slot, Enum, Link, State, Op, Param and Tr) where *Class* is the set of the model classes, *Slot* is the set of the slots, *Enum* is the set of the model enumerations, *Link* is the set of the instantiations of the model associations, *State* is the set of the states found in the state-chart, *Op* is the set of the operations defined in the class diagram, *Param* is the set of all parameters of the operations and *Tr* is the set of the UML4MBT transitions defined in the state-chart. Notice that a slot is the instantiation of the attribute of a class.

Moreover, we introduce the notation *inst(C)* that means 'all instances of class C', *enum(E)* that means 'all literal values in the enumeration E', *pre(X)* that returns the precondition of the operation X or the guard of the UML4MBT transition X, *post(X)* that returns the postcondition of the operation X or the action of the UML4MBT transition X, *param(O)* that returns the parameters of the operation O, *in(T)* that returns the source state of the UML4MBT transition T and *out(T)* that returns the target state of the UML4MBT transition T.

The SMT model functions are noted with fun_{cat}(*name:type*) where

- *fun* can take the values var for a variable, *svar* for a stepVariable and *con* for a constant. In the case of constants, one has the possibility to note directly the value *true*, *false* or an integer.
- *cat* stores the category, if relevant, of the function: *state*, *input*, *animation* or $test$.

- *name* is the name of the function.
- *type* is the type of the function. *type* can be *Bool* for boolean, *Int* for integer or the name of a user-defined named type. A named type is defined by *name:Named* where *name* is the name of the type. A set type is defined by *name:eltName:Set* where *name* is the name of the type and *eltName* is the name of the element's type.

For example, $svar_{state}$(*att1:Bool*) represents a boolean stepVariable, named $att1$, that belongs to the state variables.

7.2. *UML4MBT conversion*

Table I contains conversion rules concerning the static part of the model described in the class and object diagrams. The table is divided vertically into three parts by two double lines. The left part states the name of the rule. The middle part shows conditions on UML4MBT entities to trigger the rule. The right part describes the effect of the rule. A rule can create SMT entities to represent a type, a function or an assumption. All assumptions of this table belong to the 'static' category.

To illustrate how a conversion rule works, the conversion of the enumeration named 'Position' of the Robot model that contains literal 'up' and 'down' is detailed. First, the rules that have a satisfied condition are selected. In this case, the rule named 'Enum' is selected because the condition $HPosition \in Enum$ is true. The right part of the table indicates that we must create a named type 'Position', two constants named 'up' and 'down' of the 'Position' type and the assumption 'distinct up down'.

Classes are converted into named types, and *Instances* are converted into constants. Moreover, the values of the constants are differentiated with an assumption. To represent the null value, a special constant, named $ClassName_{none}$, is created.

Slots are converted into SMT functions of the correct type. In the case of range, enumeration and named types, SMT assumptions are needed to bound the value of the created functions. *Parameters* follow the same logic and only differ by the variable category; slots belong to state variables and parameters belongs to input variables. A slot of an instance can not store another instance. In this case, a link with a maximum cardinality of 1 is used.

The conversion of the *associations* differs according to their multiplicity. When there is a multiplicity of 0..1 or 1..1, a process identical to the slot conversion is followed. In the other cases, one needs to create an SMT collection to represent the association. Rules about association are only valid in the case of unidirectional association. But a bidirectional association can be replaced by two unidirectional associations and some constraints. These two rules are named 'Link-set' and 'Link-single' in the table. In these rules, $x_{a \to b}$ means the link x between the class a and b. For example, the link connecting the incoming conveyor belt to the robot, named *robotI*, is converted by the rule 'Link-single' because the maximum cardinality of the link is one and therefore creates a single variable $svar_{state}$(*robotI:Robot*) and constraint $robotI = robot$. On the contrary, the variable representing all instances of the outgoing belt class will lead to the creation of

1. The type $set_{outgoingConveyorBelt} : outgoingConveyorBelt : Set$;
2. The variable $svar_{state}(allInstance_{outgoingConveyorBelt} : set_{outgoingConveyorBelt})$;
3. The constraint
   $(has(allInstance_{outgoingConveyorBelt}, upLeftConveyorBelt) = 1$
   $\vee\ has(allInstance_{outgoingConveyorBelt}, upLeftConveyorBelt) = 0)$
   $\wedge$
   $(has(allInstance_{outgoingConveyorBelt}, upRightConveyorBelt) = 1$
   $\vee\ has(allInstance_{outgoingConveyorBelt}, upRightConveyorBelt) = 0)$.

7.3. *OCL4MBT conversion*

In this section and the following, rules are described in Tables II–IV. The left part is unchanged from the previous section. The middle part shows the formula to convert and the conditions on this formula. The right part can now have up to three columns, named Formula, Function and Assumption.

Table I. Rules for UML4MBT model.

	UML4MBT	Satisfiability Modulo Theories		
Name	Condition	Type	Function	Assumption
Slot-int	$x \in \mathbf{Slot} \wedge x \in \mathbb{N}$		$\mathbf{svar_{state}}(x : Int)$	
Slot-bool	$x \in \mathbf{Slot} \wedge x \in \{false, true\}$		$\mathbf{svar_{state}}(x : Bool)$	
Slot-range	$x \in \mathbf{Slot} \wedge x \in \mathbb{N} \wedge x_{min} \leq x \leq x_{max}$		$\mathbf{svar_{state}}(x : Int)$	$x \leq x_{max} \wedge x \geq x_{min}$
Slot-enum	$x \in \mathbf{Slot} \wedge e \in \mathbf{Enum} \wedge l \in \mathbf{enum}(e) = \{l_0, l_1, \ldots, l_n\} \wedge x = l$		$\mathbf{svar_{state}}(x : e)$	$\bigvee_{j=0}^{n} x = l_j$
Param-int	$x \in \mathbf{Param} \wedge x \in \mathbb{N}$		$\mathbf{svar_{input}}(x : Int)$	
Param-bool	$x \in \mathbf{Param} \wedge x \in \{false, true\}$		$\mathbf{svar_{input}}(x : Bool)$	
Param-range	$x \in \mathbf{Param} \wedge x \in \mathbb{N} \wedge x_{min} \leq x \leq x_{max}$		$\mathbf{svar_{input}}(x : Int)$	$x \leq x_{max} \wedge x \geq x_{min}$
Param-enum	$x \in \mathbf{Param} \wedge e \in \mathbf{Enum} \wedge l \in \mathbf{enum}(e) = \{l_0, l_1, \ldots, l_n\} \wedge x = l$		$\mathbf{svar_{input}}(x : e)$	$\bigvee_{j=0}^{n} x = l_j$
Param-inst	$x \in \mathbf{Param} \wedge c \in \mathbf{Class} \wedge i \in \mathbf{inst}(c) = \{i_0, i_1, \ldots, i_n\} \wedge x = i$		$\mathbf{svar_{input}}(x : c)$	$\bigvee_{j=0}^{n} x = i_j$
Class	$x \in \mathbf{Class} \wedge \mathbf{inst}(x) = \{i_0, i_1, \ldots, i_n\}$	$x : Named$	$\mathbf{con}(x_{none} : x)$	$distinct\ i_0, \ldots, i_n$
Instance	$x \in \mathbf{inst}(class)$		$\mathbf{con}(x : class)$	
Enum	$x \in \mathbf{Enum} \wedge l \in \mathbf{enum}(x) = \{l_0, l_1, \ldots, l_n\}$	$x : Named$	$\mathbf{con}(l : x)$	$distinct\ l_0, \ldots, l_n$
Link-single	$x = x_{a \to b} \in \mathbf{Link} \wedge max(\lvert x \rvert) = 1 \wedge \mathbf{inst}(b) = \{i_0, i_1, \ldots, i_n\}$		$\mathbf{svar_{state}}(x : b)$	$\bigvee_{j=0}^{n} x = i_j$
Link-set	$x = x_{a \to b} \in \mathbf{Link} \wedge max(\lvert x \rvert) > 1 \wedge \mathbf{inst}(b) = \{i_0, i_1, \ldots, i_n\}$	$set_x : x : Set$	$\mathbf{svar_{state}}(x : set_x)$	$\bigwedge_{j=0}^{n} has(x, i_j) = 0 \vee has(x, i_j) = 1$

Table II. Rules for OCL4MBT independent of context.

	OCL4MBT		Satisfiability Modulo Theories		
Name	Condition	Formula	Formula	Function	Assumption
Int-Access	$slot \in \mathbb{N}$	$var.slot$	$access$	$\mathbf{svar}(access : Int)$	$\bigwedge_{i=0}^{n} var = b_i \Rightarrow access = slot_{bi}$
Bool-Access	$slot \in \{false, true\}$	$var.slot$	$access$	$\mathbf{svar}(access : Bool)$	$\bigwedge_{i=0}^{n} var = b_i \Rightarrow access = slot_{bi}$
Enum-Access	$slot \in enum$	$var.slot$	$access$	$\mathbf{svar}(access : enum)$	$\bigwedge_{i=0}^{n} var = b_i \Rightarrow access = slot_{bi}$
Link-Access	$slot = slot_{a \to b} \in \mathbf{Link}$	$var.slot$	$access$	$\mathbf{svar}(access : b)$	$\bigwedge_{i=0}^{n} var = b_i \Rightarrow access = slot_{bi}$

For these rules, we have $a, b \in \mathbf{Class}$, $var \in \mathbf{inst}(b) = \{b_0, b_1, \dots, b_n\}$, and $slot_{b_i}$ is the slot of the instance b_i.

The Function and Assumption columns list the necessary additional SMT entities to create. The Formula column contains the current translation of the OCL4MBT formula. Moreover, the conversion process is recursive; if it is possible, a new rule on the formula described in the Formula column is executed.

OCL4MBT is a contextual language. Therefore, the meanings of the operators depend on the context. The rules unaffected by the context are described first. Then the rules under the assignment context are presented, and finally, the rules under the evaluation context will be introduced.

Table II contains the rules that are common to both the assignment and the evaluation contexts. For these rules, one has $a, b \in Class$, $var \in inst(b) = \{b_0, b_1, \dots, b_n\}$, and $slot_{b_i}$ is the slot of the instance b_i. The rules describe the conversion process of the access to a slot or a link from a variable representing an instance like an input parameter of an operation. The common idea behind these rules is to use the property of UML4MBT that all instances of a class are known, and therefore, it is always possible to iterate on instances belonging to a collection.

Under the assignment context, OCL4MBT is an imperative language. Therefore, it is possible to perform successive assignments on the same variable. To translate this behaviour into the SMT language, we choose to use the Static Single Assignment form that is defined by Cytron and Ferrante [16] to assure the uniqueness of each assigned function. And that is why a new SMT function has to be created for each assignment. For example, $a = 1 \wedge a = a + 1$ becomes $a_0 = 1 \wedge a_1 = a_0 + 1$. Moreover, in this context, all operators must return the true value because it is always possible to assign a new value.

During the conversion process, a UML4MBT entity can be converted into multiple SMT functions. A link between all these functions is created by introducing a phi function named φ. The behaviour of the φ depends on the context. In the assignment context, φ returns a fresh SMT function. In the evaluation context, φ returns the last employed function.

Table III contains the rules used in the assignment context. For all rules in this table, one has $c \in Class$ and $i_a \in inst(c) = \{i_0, i_1, \dots, i_n\}$. The assignment of a value to a function is straightforward. However, we highlight the rule *A-undef* that assigns a null value; it does not use a generic null but a value specific for each class. The conversion of the set operators uses two properties of the UML4MBT model: a collection only contains instances and all instances are known. Therefore, the possibilities can always be all enumerated.

The conversion of an operation call is more complex and cannot be represented directly by a rule. An operation call is composed of three parts: an operation with a *pre/post* condition, a list of input parameters and possibly an output parameter. Under the assignment context, a call must not have an output parameter. A call is used to modify the state of the system. The conversion process has four steps. The first is to determine if the call is valid, that is if the predicate representing the '*pre*' condition of the operation can be satisfied with the specified values for the input parameters. Therefore, this predicate must be inlined in the previous evaluation context of the OCL4MBT formula. In practice, this context can be the condition in the '*if then else*' operator or in the '*pre*' condition of the operation. The second step is to substitute the call by a boolean variable. By definition, this predicate must be true if the call is activated to respect the conventions of the assignment context. The

Table III. Rules for OCL4MBT in assignment context.

	OCL4MBT		Satisfiability Modulo Theories	
Name	Condition	Formula	Formula	Assumption
A-int	$in_1, in_2 \in \mathbb{N}$	$in_1 = in_2$	$\varphi(in_1) = in_2$	$\varphi(in_1)$
A-bool	$b_1, b_2 \in \{false, true\}$	$b_1 = b_2$	$\varphi(b_1) = b_2$	$\varphi(b_1)$
A-enum	$l_1 \in \mathbf{Slot} \wedge \{l_1, l_2\} \in \mathbf{enum}(e_3) \wedge e_3 \in \mathbf{Enum}$	$l_1 = l_2$	$\varphi(l_1) = l_2$	$\varphi(l_1)$
A-inst		$i_1 = i_2$	$\varphi(i_1) = i_2$	$\varphi(i_1)$
A-undef		$i_a.oclIsUndefined()$	$\varphi(i_a) = class_{none}$	$\varphi(i_a)$
A-set	$S_1 = S_2$	$S_1, S_2 \subseteq \mathbf{inst}(c)$	$\bigwedge_{i=0}^{n} \varphi\left(S_1\left(i_j\right)\right) = S_2(i_i)$	$\varphi(S_1)$
A-include	$S_1 \subseteq \mathbf{inst}(c)$	$S_1.includes\,(i_a)$	$\varphi\left(S_1(i_a)\right) = 1 \wedge \bigwedge_{j=0}^{a-1} \varphi\left(S_1(i_j)\right) = S_1(i_j) \wedge \bigwedge_{j=a+1}^{n} \varphi\left(S_1(i_j)\right) = S_1\left(i_j\right)$	$\varphi(S_1)$
A-exclude	$S_1 \subseteq \mathbf{inst}(c)$	$S_1.excludes(i_a)$	$\varphi\left(S_1(i_a)\right) = 0 \wedge \bigwedge_{j=0}^{a-1} \varphi\left(S_1(i_j)\right) = S_1(i_j) \wedge \bigwedge_{j=a+1}^{n} \varphi\left(S_1(i_j)\right) = S_1(i_j)$	$\varphi(S_1)$
A-empty	$S_1.empty()$	$S_1 \subseteq \mathbf{inst}(c)$	$\bigwedge_{j=0}^{n} \varphi\left(S_1(i_j)\right) = 0$	$\varphi(S_1)$
A-forAll	$S1.forAll(expr(i_j)) \wedge S_1 \subseteq \mathbf{inst}(c)$	$expr : c \rightarrow Bool$	$\bigwedge_{j=0}^{n} expr(i_j)$	

For all rules in this table, we have $c \in \mathbf{Class}$ and $i_a \in \mathbf{inst}(c) = \{i_0, i_1, \ldots, i_n\}$.

Table IV. Rules for OCL4MBT in evaluation context.

	UML	Satisfiability Modulo Theories		
Name	Formula	Formula	Function	Assumption
E-bool-equality	$b_1 = b_2$	$b_1 = b_2$		
E-bool-inequality	$b_1 \neq b_2$	$\neg(b_1 = b_2)$		
E-bool-or	$b_1 \vee b_2$	$b_1 \vee b_2$		
E-bool-and	$b_1 \wedge b_2$	$b_1 \wedge b_2$		
E-bool-xor	$b_1 \oplus b_2$	$(b_1 \wedge \neg b_2) \vee (\neg b_1 \wedge b_2)$		
E-bool-ite	$\textit{if}\, b_1\ \textit{then}\ b_2\ \textit{else}\ b_3$	$\textit{ite}\ b_1\ \Phi(b_2)\ \Phi(b_3)$		
E-bool-false	*false*	*false*		
E-bool-true	*true*	*true*		
E-bool-var	b_1	$\varphi(b_1)$		
E-int-equality	$in_1 = in_2$	$in_1 = in_2$		
E-int-inequality	$in_1 \neq in_2$	$\neg\,(in_1 = in_2)$		
E-int-strict-inf	$in_1 < in_2$	$in_1 < in_2$		
E-int-inferior	$in_1 \leqslant in_2$	$in_1 \leqslant in_2$		
E-int-strict-sup	$in_1 \geq in_2$	$in_1 \geq in_2$		
E-int-superior	$in_1 > in_2$	$in_1 > in_2$		
E-int-plus	$in_1 + in_2$	$in_1 + in_2$		
E-int-minus	$in_1 - in_2$	$in_1 - in_2$		
E-int-mult	$in_1 * in_2$	$in_1 * in_2$		
E-int-minus-1	$-in_1$	$-in_1$		
E-int-div	$in_1.div(in_2)$	$div\ in_1\ in_2$		
E-int-abs	$in_1.abs()$	in_2	$\mathbf{svar}(in_2 : Int)$	$ite\ (in_1 < 0)\ (in_2 = (-in_1))\ (in_2 = in_1)$
E-int-max	$in_1.max(in_2)$	in_3	$\mathbf{svar}(in_3 : Int)$	$ite\ (in_1 < in_2)\ (in_3 = in_2)\ (in_3 = in_1)$
E-int-min	$in_1.min(in_2)$	in_3	$\mathbf{svar}(in_3 : Int)$	$ite\ (in_1 > in_2)\ (in_3 = in_2)\ (in_3 = in_1)$
E-int-number	$in_1 \notin \mathbf{Slot}$	in_1		
E-int-slot	$in_1 \in \mathbf{Slot}$	$\varphi(in_1)$		
E-enum-equality	$l_1 = l_2$	$l_1 = l_2$		
E-enum-inequality	$l_1 \neq l_2$	$\neg\,(l_1 = l_2)$		
E-enum-value	$l_1 \notin \mathbf{Slot}$	l_1		
E-enum-slot	$l_1 \in \mathbf{Slot}$	$\varphi(l_1)$		

Table IV. *(Continued)*.

E-inst-equality	$i_1 = i_2$	$i_1 = i_2$		
E-inst-inequality	$i_1 \neq i_2$	$\neg (i_1 = i_2)$		
E-inst-undefined	i_1.*oclisUndefined*()	$i_1 = c_{none}$		
E-inst-value	$i_1 \notin$ **Slot**	i_1		
E-inst-value	$i_1 \in$ **Slot**	$\varphi(i_1)$		
E-inst-allInstance	*c.allInstance*()	c_{all}	$\mathbf{con}(c_{all} : set_c)$	$\bigwedge_{j=0}^{n} c_{all}(i_j) = 1$
ES-equal	$S_1 = S_2$	$\bigwedge_{j=0}^{n} S_1(i_j) = S_2(i_j)$		
ES-inequal	$S_1 \neq S_2$	$\neg \left(\bigwedge_{j=0}^{n} S_1(i_j) = S_2(i_j)\right)$		
ES-size	S_1.*size*()	$\sum_{j=0}^{n} S_1(i_j)$		
ES-include	S_1.*include*(i)	*ite* $(S_1(i) = 1)$ *true false*		
ES-exclude	S_1.*exclude*(i)	*ite* $(S_1(i) = 0)$ *true false*		
ES-inclAll	S_1.*includeAll*(S_2)	$\bigwedge_{j=0}^{n} (S_2(i_j) = 1)$ $\rightarrow (S_1(i_j) = 1)$		
ES-exclAll	S_1.*excludeAll*(S_2)	$\bigwedge_{j=0}^{n} (S_2(i_j) = 1)$ $\rightarrow (S_1(i_j) = 0)$		
ES-empty	S_1.*isEmpty*()	$\sum_{j=0}^{n} S_1(i_j) = 0$		
ES-notEmp	S_1.*notEmpty*()	$\neg \left(\sum_{j=0}^{n} S_1(i_j) = 0\right)$		
ES-union	S_1.*union*$(S2)$	S_3	$\mathbf{svar}(S_3 : set_c)$	$\bigwedge_{j=0}^{n} ite(S_2(i_j) + S_1(i_j) = 0)$ $(S_3(i_j) = 0)(S_3(i_j) = 1)$
ES-inter	S_1.*inter*$(S2)$	S_3	$\mathbf{svar}(S_3 : set_c)$	$\bigwedge_{j=0}^{n} ite(S_2(i_j) + S_1(i_j) = 2)$ $(S_3(i_j) = 1)(S_3(i_j) = 0)$
ES-exist	S_1.*exist*$(expr(i))$	$\bigvee_{j=0}^{n} expr(i_j)$		
ES-forAll	S_1.*forAll*$(expr(i))$	$\bigwedge_{j=0}^{n} expr(i_j)$		
ES-any	S_1.*any*$(expr(i))$	i_{any}	$\mathbf{svar}(i_{any} : c)$	$iteexpr(i_0)(i_{any} = i_0)$ $(iteexpr(i_1)(i_{any} = i_1)$ $(\ldots iteexpr(i_n)(i_{any} = i_n)$ $(i_{any} = i_{none}))\ldots)$
ES-value	S_1	$\varphi(S_1)$		

In this table, we have $b_1, b_2, b_3 \in \{false, true\}$, $in_1, in_2, in_3 \in \mathbb{N}$, $e \in$ **Enum**, $l_1, l_2 \in \mathbf{enum}(e)$, $c \in$ **Class**, $\{i_0, i_1, \ldots, i_n\} = \mathbf{inst}(c)$ and $S_1, S_2, S_3 \subseteq \mathbf{inst}(c)$.

third step is to create a constraint that implies the '*post*' condition of the operation if the boolean variable is true. The last step is to substitute the input parameters by their value.

Under the evaluation context, OCL4MBT respects the standard semantics of the OCL language. Table IV contains the rules used in the evaluation context. In this table, one has $b_1, b_2, b_3 \in \{false, true\}$, $in_1, in_2, in_3 \in \mathbb{N}$, $e \in Enum$, $l_1, l_2 \in Enum(e)$, $c \in Class$, $\{i_0, i_1, \ldots, i_n\} = Inst(c)$ and $S_1, S_2, S_3 \subseteq Inst(c)$.

Three problems will be highlighted. The first appears in the 'if then else' operator. This operator can affect the state of the model differently in function of the branch selected. For example, the formula $if\ b\ then\ b = false\ else\ true$ becomes in the single state assignment form $if\ b_0\ then\ b_1 = false\ else\ true$. Therefore, $\varphi(b)$ returns a different value for each branch. In the 'then' branch, there is $\varphi(b) = b_1$. In the 'else' branch, there is $\varphi(b) = b_0$. The standard solution defined by Cytron *et al.* [16] is to unify the result through a Φ function. In the example, it gives *ite* $b_0\ (b_1 = false)(b_1 = b_0 \wedge true)$.

A second problem is to convert the set operators. The SMT metamodel has only the 'has' operator corresponding to a membership function. This function returns 1 if the element is present in the set. With the properties of the UML4MBT language, one can always enumerate all instances. Therefore, the cardinality, defined in the rule 'ES-size', is the sum of the membership function applied to all potential instances of the set. This operator can also be used to define, in the same way, the union and intersection operators.

The third and last problem is the conversion of an operation call under the evaluation context. This conversion cannot be directly represented by a rule. In this case, OCL4MBT has the restriction that an operation call must not modify the state of the system. The conversion has four parts. Like in the assignment context, the '*pre*' condition of the operation called is inlined in the previous evaluation context and the input parameters are replaced by their values. The result parameter is represented by a new variable of the corresponding type. This parameter can be an integer, an enumeration literal or an instance. Finally, a new constraint representing the '*post*' condition of the operation called must be created.

7.4. Animation rules

As it has been presented in Section 5.2, the animation of the system is represented as a transition system. In this case, an operation described in the class diagram or a UML4MBT transition found in the state-chart is considered as a transition between two states of the system. Therefore, an animation is a finite sequence of transitions. The execution of a transition is called a step. The constraint that two transitions cannot be executed simultaneously is also added.

Table V contains rules to animate the model. For all rules in this table, one has $a_0, a_1, \ldots, a_n = var(x)$ and the word *next* under a stepVariable indicates that the value of this function is considered after the execution of the transition. Moreover, all assumptions described in the table belong to the *animation* category.

The rule *State* creates a stepVariable to represent each state found in the UML4MBT state-chart.

The rule *Operation* associates three functions for each operation. The stepVariable x represents the current operation. trig_x is a flag memorizing the activation of an operation during the last step. This flag is principally used to represent the call to an operation during the activation of a state-chart transition. $step_x$ is a flag that is true if this operation is executed during the current step. The assumption created by this rule can be read as follows: if the precondition of the operation x is verified and this operation is executed during this step, then its activation is memorized for the next step, the postcondition is executed and new values are assigned to state variables, else the nonactivation of the operation is memorized.

The rule *Transition* shares a common pattern with the previous rule. The main differences are that we memorize when a UML4MBT transition of the state-chart has been executed through the $\text{trig}_{none_{next}} = true$ part of the assumption and the condition is modified to verify that the model is in the correct state of the state-chart and the right operation has been previously executed through the $trig(x) \wedge in(x) = state$ part of the assumption.

Table V. Rules for the model animation.

UML		Satisfiability Modulo Theories		
Name	Condition	Sort	Function	Assumption
State	$x \in \mathbf{State}$		$\mathbf{svar}_{\mathbf{animation}}(x : \mathit{State})$	
Operation	$x \in \mathbf{Ops}$		$\mathbf{svar}_{\mathbf{animation}}(x : \mathit{Event})$	$\mathit{ite}\ (x = \mathit{event} \wedge \mathbf{pre}(x))$
			$\mathbf{svar}_{\mathbf{animation}}(\mathrm{trig}_x : \mathit{Bool})$	$(\mathrm{trig}_{x_{next}} = \mathit{true} \wedge \mathbf{post}(x) \wedge \bigwedge_{i=0}^{n} a_{i_{next}} = \varphi(a_i))$
			$\mathbf{svar}_{\mathbf{animation}}(\mathit{step}_x : \mathit{Bool})$	$(\mathit{step}_x = \mathit{false})$
Transition	$x \in \mathbf{Trs}$		$\mathbf{svar}_{\mathbf{animation}}(x : \mathit{Event})$	$\mathit{ite}\ (x = \mathit{event} \wedge \mathbf{pre}(x) \wedge \mathbf{trig}(x) \wedge \mathbf{in}(x) = \mathit{state})$
			$\mathbf{svar}_{\mathbf{animation}}(\mathit{step}_x : \mathit{Bool})$	$\big(\mathrm{trig}_{none_{next}} = \mathit{true} \wedge \mathit{state}_{next} = \mathbf{out}(x) \wedge \bigwedge_{i=0}^{n} a_{i_{next}} = \varphi(a_i)\big)$
				$(\mathit{step}_x = \mathit{false})$
Model		$\mathit{Event} : \mathit{Named}$	$\mathbf{svar}_{\mathbf{animation}}(\mathit{event} : \mathit{Event})$	$\mathit{distinct}\ tr_0, \ldots, tr_n, op_0, \ldots, op_p$
		$\mathit{State} : \mathit{Named}$	$\mathbf{con}_{\mathbf{animation}}(op_{none} : \mathit{Event})$	$\bigvee_{i=0}^{n} \mathit{step}_{tr_i} \vee \bigvee_{i=0}^{p} \mathit{step}_{op_i} \vee \mathit{step}_{none}$
			$\mathbf{con}_{\mathbf{animation}}(\mathrm{trig}_{none} : \mathit{Bool})$	$\mathit{ite}\ (x = op_{none})\ \big(\mathrm{trig}_{none_{next}} = \mathit{true} \wedge \bigwedge_{i=0}^{n} a_{i_{next}} = a_i\big)\ (\mathit{step}_{none} = \mathit{false})$
			$\mathbf{svar}_{\mathbf{animation}}(\mathit{state} : \mathit{State})$	
			$\mathbf{con}_{\mathbf{animation}}(\mathit{step}_{none} : \mathit{Bool})$	

For all rules in this table, we have $a_0, a_1, \ldots, a_n = \mathbf{var}(x)$ and the word *next* under a stepVariable indicates that we consider the value of this function after the execution of the transition.

The rule *Model* fulfils two objectives. The first is to force the solver to execute a sequence of transitions. The rule creates a lock function *event* that prevents two or more transitions to be executed simultaneously and an assumption on the stepvariables $step_x$ that forces to execute at least one transition. The second objective is to create a dummy transition without effect on the system. In order to write an SMT model into an SMT-LIB instance, the length of the animation sequence must be selected before the resolution by the SMT solver. Therefore, to remove the possibility of obtaining an unsatisfiable benchmark because the sequence of animation reached a final state of the system with a shorter sequence of transitions than the desired length, a dummy transition is introduced.

8. STRATEGIES

The goal of the strategies is to reduce the time needed to reach the targets without sacrificing the test coverage. These strategies are necessary because there is a gap in the conceptual level between our goal (generating tests) and our method (solving a first-order logic formula). Therefore, the strategies make the link and transform the resolution of a formula into the animation of the model to generate tests.

The creation of the strategies uses three facts. First, many targets can be found during each animation of the model with the SMT solver. Second, the resolution time of the solver grows exponentially with the animation's length. Third, the resolution time increases when the number of assumptions and functions increases. Using these facts, the generation time can be reduced with adapted strategies.

To illustrate the strategies, Figure 4 shows the test generation process on an example. The goal is to generate tests to cover four targets, named A, B, C and D. These targets can be reached with transition sequences of 3, 5, 7 and 11 steps, respectively. Moreover, targets B and C cannot be found during the same animation. Finally, the example has three operations named OpA, OpB and OpC.

In Figure 4, the length of the animations is given on the vertical scale. A node represents a state of the system. The number in the nodes gives the walk-order, and the letter is the test target to reach. An arc represents an animation. If the arrow is drawn with a dotted line, the animation was unsuccessful. However, we do not know if the animation fails because the target is always unreachable or because the length of the animation is too short. Therefore, the corresponding test targets were not reached. A continuous line means that the animation was successful and a sequence of transitions was found to reach the corresponding test target. To simplify the figure, the strategies use only one thread. However, all strategies can be parallelized.

8.1. Basic strategy

The basic strategy defines a direct approach useful to study the influence of the length of the animation, the number of assumptions and the satisfiability of the problem. In this case, the solver must solve a problem for each test target and the length of the transition sequences is equal to the value given by a parameter.

In Figure 4, this strategy needs to execute four animations of 12 steps to reach the targets A, B, C and D. The length has been chosen to cover all test targets.

8.2. Step-increase strategy

The key ideas behind this strategy are to minimize the number of SMT instances to solve and to prefer the shortest animations. The strategy increases progressively the length of the animation by a fixed number, named 'increase', and searches in parallel different test targets. To ensure the termination, the length of the animations are bound with a parameter named 'maximum length'. However, when the length of the animations is reduced, the probability to reach a test target is also reduced. In this strategy, the initial state is always the state defined by the UML4MBT object diagram.

As illustrated on Figure 4, the strategy starts by executing four animations of four steps to reach each target. However, the animations searching B, C and D fail because their length is too short to

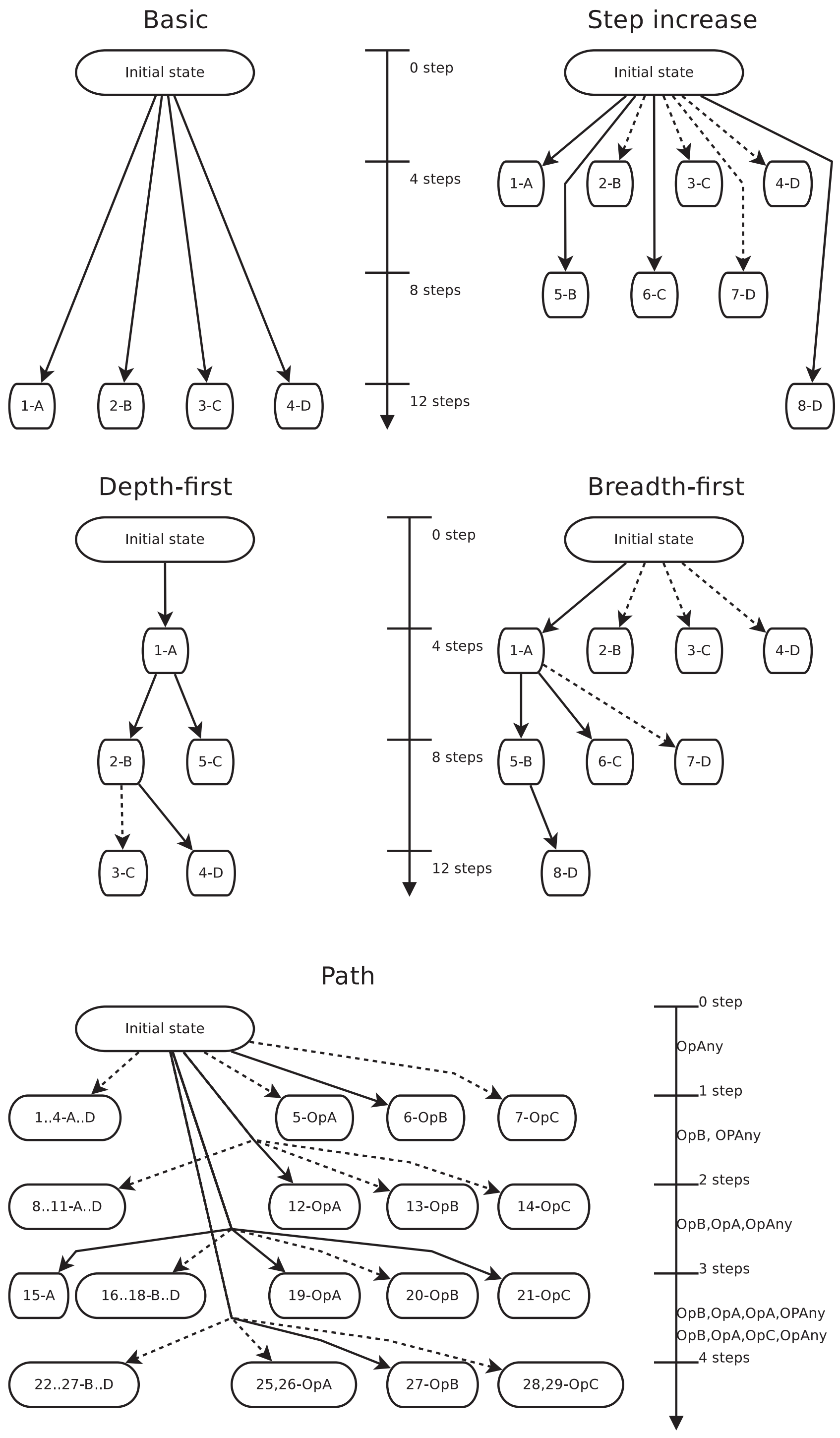


Figure 4. Test generation strategies.

reach the targets. Therefore, the length of the animations is increased to eight steps and the process restart. In the second batch of animations, targets B and C are reached but target D is not covered. To find the last target, the length is increased again. Finally, target D is reached with an animation of 12 steps.

Table VI. Animations executed during the *step-increase* strategy on the Robot.

Number	Length	Initial state	Targets
1	3	ρ	$Cpt_{nextItem}, Cpt_{evacuate}, Cpt_{grab}, Cpt_{drop.-R}, Cpt_{drop.-L}$ $Cpt_{rot.-R}, Cpt_{rot.-L}, Cpt_{transl.-U}, Cpt_{transl.-D}$
2	3	ρ	$Cpt_{evacuate}, Cpt_{drop.-R}, Cpt_{drop.-L}$ $Cpt_{rot.-R}, Cpt_{rot.-L}, Cpt_{transl.-D}$
3	5	ρ	$Cpt_{evacuate}, Cpt_{drop.-R}, Cpt_{drop.-L}$ $Cpt_{rot.-R}, Cpt_{rot.-L}, Cpt_{transl.-D}$
4	5	ρ	$Cpt_{drop.-R}, Cpt_{rot.-R}, Cpt_{rot.-L}, Cpt_{transl.-D}$
5	5	ρ	$Cpt_{rot.-L}, Cpt_{transl.-D}$
6	7	ρ	$Cpt_{rot.-L}, Cpt_{transl.-D}$

Table VII. Tests created during the *step-increase* strategy on the Robot.

Animations	Tests
1	targets: $Cpt_{nextItem}, Cpt_{grab}, Cpt_{transl.-U}$ sequence: nextItem(T1), grab(), translate(up)
3	targets: $Cpt_{drop.-L}, Cpt_{evacuate}$ sequence: nextItem(T1), grab(), translate(up), drop(), evacuate()
4	targets: $Cpt_{rot.-R}, Cpt_{drop.-R}$ sequence: nextItem(T2), grab(), translate(up), rotate(right), drop()
6	targets: $Cpt_{rot.-L}, Cpt_{transl.-D}$ sequence: nextItem(T2), grab(), translate(up), rotate(right), drop(), rot.(gauche), translate(down)

Now we want to apply this strategy on the running example. To cover the elementary behaviours, nine test targets have been created, one for each operation except for 'translate', 'rotate' and 'drop' that have two targets: $Cpt_{nextItem}$, $Cpt_{rot.-L}$ (rotation to left), $Cpt_{rot.-R}$ (rotation to right), $Cpt_{trans.-U}$ (up translation), $Cpt_{trans.-D}$ (down translation), Cpt_{grab}, $Cpt_{drop.-L}$, $Cpt_{drop.-R}$ and $Cpt_{evacuate}$. The strategy has the following configuration: the maximum length of the animations is nine steps, the initial length is three steps and the incremental length is 2.

Tables VI and VII detail the test generation process. Table VI lists the animations. The first column is the order of execution. The second gives the length. The third indicates the initial state. ρ is the state defined by the object diagram. The last column is the test goal. In this case, an animation must reach at least one target in this list. Table VII gives the tests created by the animation of Table VI. A test is described by a sequence of operations and a list of the reached test targets. The number in the first column corresponds to the number of the first column in Table VI.

8.3. Depth-first strategy and breadth-first strategy

The depth-first and breadth-first strategies combine the possibility to divide a transition sequence into a list of shorter sequences and the fact that the resolution time does not scale linearly when the length of the sequence increases. The idea is to execute animations with a fixed length and to change the initial state of the system. The initial state is modified to correspond to the state of the system after a previous animation. The strategy explores the search space with a depth-first or breadth-first algorithm.

Therefore, these strategies are configurable through two parameters: one is the length of the animations and the other is the maximum number of successive animations. This parameter ensures the termination. Two animations are successive if the final state of the first animation is the initial state of the second animation.

An example is given in Figure 4. In the case of a depth-first exploration, the strategy begins to execute an animation of four steps to reach the target A. Then the next animation of four steps tries

Table VIII. Animations executed during the *depth-first* strategy on the Robot.

Num.	Length	Initial state	Targets
1	3	ρ	$Cpt_{nextItem}, Cpt_{evacuate}, Cpt_{grab}, Cpt_{drop.-R}, Cpt_{drop.-L}$ $Cpt_{rot.-R}, Cpt_{rot.-L}, Cpt_{transl.-U}, Cpt_{transl.-D}$
2	3	1	$Cpt_{evacuate}, Cpt_{drop.-R}, Cpt_{drop.-L}$ $Cpt_{rot.-R}, Cpt_{rot.-L}, Cpt_{transl.-D}$
3	3	2	$Cpt_{evacuate}, Cpt_{drop.-R}, Cpt_{rot.-R}, Cpt_{rot.-L}$
4	3	2	$Cpt_{evacuate}, Cpt_{rot.-L}, Cpt_{drop.-R}$
5	3	2	$Cpt_{rot.-L}, Cpt_{drop.-R}$
6	3	1	$Cpt_{rot.-L}, Cpt_{drop.-R}$
7	3	ρ	$Cpt_{rot.-L}, Cpt_{drop.-R}$

Table IX. Tests created during the *depth-first* strategy on the Robot.

Animations	Tests
1	targets: $Cpt_{nextItem}, Cpt_{grab}, Cpt_{transl.-U}$ sequence: nextItem(T1), grab(), translate(up)
2	targets: $Cpt_{drop.-L}, Cpt_{transl.-D}$ sequence: drop(), translate(down), nextItem(T2)
3	targets: $Cpt_{rot.-R}$ sequence: grab(), translate(up), rotate(right)
4	targets: $Cpt_{evacuate}$ sequence: evacuate()

to reach the target B. However, the initial step of this animation is modified to correspond to the final state of the previous animation. Then this process goes on until all targets are found or the allowed number of successive animations is exceeded.

This strategy has one disadvantage. When an animation is cut, information about the reachability is lost. When an animation of X steps is cut into two in order to find a target 'A' and the solver cannot found a solution, it is not possible to conclude that the target 'A' is not reachable in X steps. Indeed, when the animation is run for the second time, it starts from a specific state that might be incompatible with the target to reach.

In the case of a breadth-first exploration, the efficiency is greatly influenced by the deviation of the minimum lengths of the animations required to find the test targets. For example, if one executes successive animations of three steps and the majority of the targets are found around two or 10 steps, the animations with a cumulative length of six and nine steps will not succeed.

Next, the depth-first strategy is applied on the running example with animations of three steps and a maximum of three successive animations. The results are given in two tables respecting the formalism introduced in the previous strategy. Table VIII lists the animations executed during the tests generation process. The initial state of the animations can be ρ, the state defined in the object diagram or a state reached during a previous successful animation, referred by a number. Table IX shows the constructed tests.

8.4. Path strategy

The size of the SMT instance is of course dependent on the number of the operations and the transitions in the model. The key idea of the strategy is to minimize the number of the assumptions and the functions in the SMT instances to solve. The strategy is based on construction of the patterns and of two phases. Pattern catches sequences of compatible transitions. The first phase tries to reach targets with current patterns. The second phase extends patterns with all compatible operations.

To ensure the termination of the strategy, a parameter fixes a maximum length for the animations.

Table X. Animations executed during the *path* strategy on the Robot.

Num.	Length	Initial state	Constraints
1	1	ρ	Targets: $Cpt_{nextItem}, Cpt_{evacuate}, Cpt_{grab}, Cpt_{drop.-R}, Cpt_{drop.-L}, Cpt_{rot.-R}, Cpt_{rot.-L}, Cpt_{transl.-U}, Cpt_{transl.-D}$ sequence: $opAny$
2	1	ρ	Targets: $Cpt_{evacuate}, Cpt_{grab}, Cpt_{drop.-R}, Cpt_{drop.-L}, Cpt_{rot.-R}, Cpt_{rot.-L}, Cpt_{transl.-U}, Cpt_{transl.-D}$ sequence: $opAny$
3	2	ρ	Targets: $Cpt_{evacuate}, Cpt_{grab}, Cpt_{drop.-R}, Cpt_{drop.-L}, Cpt_{rot.-R}, Cpt_{rot.-L}, Cpt_{transl.-U}, Cpt_{transl.-D}$ sequence: *nextItem*(), *opAny*
4	2	ρ	Targets: $Cpt_{evacuate}, Cpt_{drop.-R}, Cpt_{drop.-L}, Cpt_{rot.-R}, Cpt_{rot.-L}, Cpt_{transl.-U}, Cpt_{transl.-D}$ sequence: *nextItem*(), *opAny*
5	3	ρ	Targets: $Cpt_{evacuate}, Cpt_{drop.-R}, Cpt_{drop.-L}, Cpt_{rot.-R}, Cpt_{rot.-L}, Cpt_{transl.-U}, Cpt_{transl.-D}$ sequence: *nextItem*(), *grab*(), *opAny*
6	3	ρ	Targets: $Cpt_{evacuate}, Cpt_{drop.-R}, Cpt_{drop.-L}, Cpt_{rot.-R}, Cpt_{rot.-L}, Cpt_{transl.-D}$ sequence: *nextItem*(), *grab*(), *opAny*
7	4	ρ	Targets: $Cpt_{evacuate}, Cpt_{drop.-R}, Cpt_{drop.-L}, Cpt_{rot.-R}, Cpt_{rot.-L}, Cpt_{transl.-D}$ sequence: *nextItem*(), *grab*(), *translate*(), *opAny*
8	4	ρ	Targets: $Cpt_{evacuate}, Cpt_{drop.-R}, Cpt_{rot.-R}, Cpt_{rot.-L}, Cpt_{transl.-D}$ sequence: *nextItem*(), *grab*(), *translate*(), *opAny*
9	4	ρ	Targets: $Cpt_{evacuate}, Cpt_{drop.-R}, Cpt_{rot.-L}, Cpt_{transl.-D}$ sequence: *nextItem*(), *grab*(), *translate*(), *opAny*
10	5	ρ	Targets: $Cpt_{evacuate}, Cpt_{drop.-R}, Cpt_{rot.-L}, Cpt_{transl.-D}$ sequence: *nextItem*(), *grab*(), *translate*(), *drop*(), *opAny*
11	5	ρ	Targets: $Cpt_{evacuate}, Cpt_{drop.-R}, Cpt_{rot.-L}$ sequence: *nextItem*(), *grab*(), *translate*(), *drop*(), *opAny*
12	5	ρ	Targets: $Cpt_{drop.-R}, Cpt_{rot.-L}$ sequence: *nextItem*(), *grab*(), *translate*(), *drop*(), *opAny*
13	5	ρ	Targets: $Cpt_{drop.-R}, Cpt_{rot.-L}$ sequence: *nextItem*(), *grab*(), *translate*(), *rotate*(), *opAny*
14	5	ρ	Targets: $Cpt_{rot.-L}$ sequence: *nextItem*(), *grab*(), *translate*(), *rotate*(), *opAny*

Figure 4 gives an example of test generation with this strategy. In this case, the length of the animations is limited to four steps. The process alternates two phases. Phase 1: The strategy tries to reach the test targets with the available patterns. At the beginning of the process, only the pattern $opAny$ (i.e. all operations are authorized) is available. In the example, the strategy executes four animations to reach the targets. All animations fail. Phase 2: the strategy discovers new patterns. To create a new pattern, an operation is substituted to $opAny$ in the previous patterns. If this pattern results in a successful animation, a new valid pattern is created by appending $opAny$. In the example, there are three operations. Therefore, three animations are executed to determine if the transitions opA, opB and opC can be executed from the initial state of the system. Only the second animation is successful. Consequently, the new pattern opB, $OpAny$ is created. Then the process restarts at phase 1.

This strategy is applied on the running example with a maximum length of five steps for the animations. Table X lists the animations executed during the tests generation process. The fourth column indicates which targets are not yet reached and which patterns are available. Table XI gives the tests built by the strategy.

Table XI. Tests created during the *path* strategy on the Robot.

Animation	Tests
1	targets: $Cpt_{nextItem}$ sequence: nextItem(T1)
3	targets: Cpt_{grab} sequence: nextItem(T1), grab()
5	targets: $Cpt_{transl.-U}$ sequence: nextItem(T1), grab(), translate(up)
7	targets: Cpt_{drop-L} sequence: nextItem(T1), grab(), translate(up), drop()
8	targets: $Cpt_{rot.-R}$ sequence: nextItem(T2), grab(), translate(up), rotate(right)
10	targets: $Cpt_{transl.-D}$ sequence: nextItem(T1), grab(), translate(up), drop(), translate(down)
11	targets: $Cpt_{evacuate}$ sequence: nextItem(T1), grab(), translate(up), drop(), evacuate()
13	targets: Cpt_{drop-R} sequence: nextItem(T2), grab(), translate(up), rotate(left), drop()

9. EXPERIMENTS

In this section, the efficiency of the method to generate tests with an SMT solver will be studied. The first part will present the models of the systems used during the experiments. Then the experiments will be shown on the different test generation strategies. Finally, the results will be analyzed and some improvements will be proposed.

9.1. Models

The experiments will be conducted on the running example (i.e. the Robot model) and the following models:

UsWires describes a system to transfer money between different bank accounts. A user must indicate the necessary information (account number, bank, amount, etc) and select an occurrence and a date to allow the bank to verify the validity of the transaction.

eCinema is a system to purchase cinema tickets through a website. To buy a ticket, a user must be logged in the database. The system tracks the different sales and can offer several promotions.

Global Platform is a smart card application management specification. The model focuses on the card life cycle. Current implementations concern the financial, mobile telecommunications, government initiatives, healthcare, retail merchants and transit domains.

ServiDirect is a website allowing a user to subscribe to different insurance packages. The model of this website has been created during the TestIndus project (http://lifc.univ-fcomte.fr/test_indus/). On the Servidirect website, a user follows a standard procedure. First, a user must provide sufficient information to allow the website to decide if the society accepts the contract. A user must choose an insurance package, input contact information and family information, and submit a medical assessment. At the end, the user and the website must agree on a common contract.

PID represents the creation, suppression and control of the processes spawned by an operating system.

Table XII gives the major characteristics of the models. Three models–eCinema, ServiDirect and PID–use a state-chart and a class diagram to describe the behaviours of the system. The other models use a class diagram and OCL4MBT description of their operations.

Table XII. Characteristics of the models.

	UsWires	eCinema	Global Platform	ServiDirect	Robot	PID
Class	34	1	12	9	3	2
Attribute	355	6	197	38	5	1
Enumeration	191	5	48	14	3	3
Association	54	13	76	14	3	3
Instance	45	14	117	26	4	5
Slot	492	14	1384	70	6	4
Operation	332	13	61	45	8	4
Transition	N/A	17	N/A	42	N/A	13

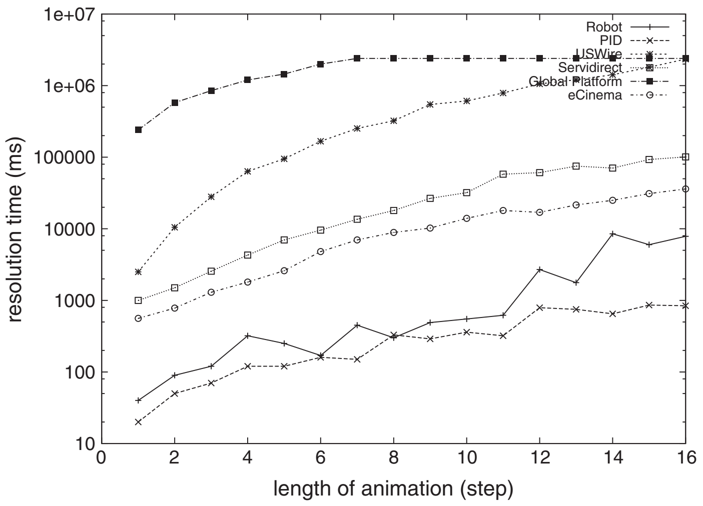
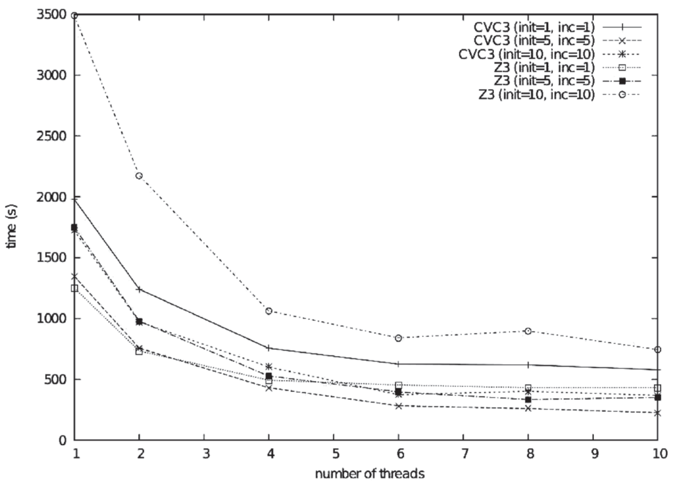


Figure 5. Resolution time in function of the transitions sequence length on several models with the Z3 solver.

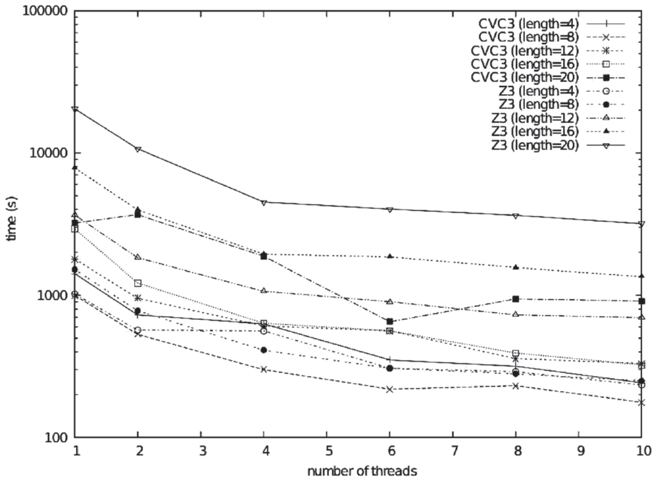
9.2. *Results*

This section presents the results of the different experiments conducted on various models. The first experiment studies the influence of the length of the animation on the resolution time of the solver. The measured time is the resolution time of the solver to find any valid sequence transition without the need to reach a target. From our modelling choice, the number of SMT functions and assumptions is linearly proportional to the length of the animation. Figure 5 shows the results. The timeout is set at 2.4×10^6 m s. The scale of the time is logarithmic. This graph confirms that two animations of $X/2$ steps are quicker than an animation of X steps. The figure proposes models with different levels of complexity and so gives an indication of the scalability of the process. The order of the curves from the top to the bottom is the same that the level of complexity of the models: PID, Robot, eCinema, Servidirect, UsWires and Global Platform. If the resolution time generally increases when the length of the animation increases, the exceptions can be explained by how an SMT solver works. Indeed the solver processes the assumptions in the SMT instance in a random order. And the order is a factor for the resolution time. For the Global Platform model, animations with greater length than 6 reach the timeout so that the curve becomes flat.

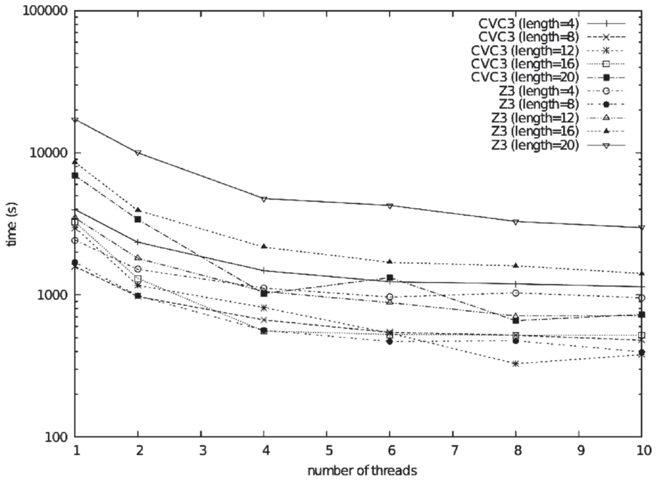
Figures 6(a)–6(c) show, respectively, the performance of the multithreaded strategies step increase, depth first and breadth first. The computations have been performed on the supercomputer facilities of the Mésocentre de Calcul de Franche-Comté. The supercomputer offers 68 computing nodes. Each node has two Intel Xeon Nehalem processors with four cores each. In the step increase strategy, the parameter *init* indicates the length of the initial animation and the length of the next animation is increased by *inc*. In the depth-first and breadth-first strategies, the *length* parameter indicates the length of the different animations. In this case, the tests cover the execution of all



(a) Results of the Step increase strategy on the Service direct model



(b) Results of the Depth-first strategy on the Service direct model



(c) Results of the Breadth-first strategy on the Service direct model

Figure 6. Results of the different multithreaded strategies.

Table XIII. Best times of the multithreaded strategies in second.

	UsWires	eCinema	Global Platform
Step increase	142574	261	8497
Depth first	15920	180	7254
Breadth first	40842	290	10524

Table XIV. Comparison of the Path and Step increase strategies (time in seconds).

	UsWire	eCinema	ServiDirect
Path	$1.592 * 10^4$	$7.121 * 10^1$	$8.6 * 10^2$
Increase 1-1	$1.385 * 10^5$	$4.100 * 10^2$	$1.979 * 10^3$
Increase 4-2	$1.447 * 10^5$	$2.975 * 10^2$	$1.875 * 10^3$
Increase 4-4	$1.576 * 10^5$	$3.846 * 10^2$	$1.346 * 10^3$
Increase 4-6	$1.830 * 10^5$	$2.158 * 10^2$	$1.725 * 10^3$

operations of the UML4MBT class diagram and all UML4MBT transitions of the state-chart. All test targets are reachable. The gain brought by the multithreading diminishes for each thread. During the animation, the targets which have not been defined in the test goal can be reached. In this case, all threads searching one of those targets will be interrupted. Therefore, the probability to be interrupted increases when the number of threads increases.

Table XIII shows the best performance for each multithreaded strategy on the most complex models. There is common tendency on almost all models: the best strategy is depth first and the worst is breadth first. This behaviour can be explained by the type of the models. For all models, there is a limited number of valid outgoing transitions (at most five) at each state. The only model that is different is the UsWires model. In this model, the verification process during the transfer of the money between two bank accounts is complex, and therefore, the tests validating this behaviour need longer animations. This difference explains the poor result of the step increase strategy.

Table XIV compares the step increase and the path strategy using a single thread and the Z3 solver. In this case, all elementary behaviours are covered. This coverage is necessary to use the path strategy. On the different models, this strategy is more efficient than the step-increase strategy by an order of magnitude.

9.3. *Analysis*

On the UsWires model, the generation time is the worst with the step increase strategy. On average, it needs an animation of 16 steps to reach a test target concerning the verification of the transfer process. And a single animation of 16 steps takes around 2140 s. This explains why the standard trend is not respected and the breadth-first strategy is better than the step increase strategy. The good performance of the depth-first strategy can be explained by the nature of the model. Each verification is represented by an operation in the class diagram like 'checkAmount' or 'checkAccountNumber'. Therefore, longer animations are needed. Moreover, there is a limited number of valid animations because if a single verification fails, the system reaches a final state.

On the Global Platform model, the performance of the different strategies is much closer than in the other models. This can be explained by the choice made during the modelling to describe an initial state of system in the object diagram that represents an active system. For example, to execute a behaviour corresponding to the activation of a module, a list of operations like the creation of the module and the registration of the module has to be executed before. A solution to obtain shorter animations is to describe an initial state of the system where some modules are already created and registered. The conclusion from these results is that the difference between the different strategies decreases when the length of the animations decreases.

On both the eCinema and the Servidirect models, the obtained result are comparable. Indeed these models share many aspects. The animation corresponds to the same workflow: buying a product on a website. The states of the state-chart are the pages of the website and the UML4MBT transitions and operations allow the navigation by a user on the website. Finally, the animations respect the same process: log in the system, fill in a form and pay for the products.

The comparison of the average time of the different strategies is not pertinent because the models and the parameters of the strategies are so different. However, some trends can be established. The path strategy is more efficient when the operations and the UML4MBT transitions of the model represent elementary behaviours and each state of the transition system has a limited number of outgoing transitions. Moreover, the test coverage must cover the elementary behaviours in order to reach all possible test targets. The efficiency of the breadth-first strategy is mostly influenced by the deviation of the minimum lengths of the animations required to found the test targets. The depth-first strategy works better when there are a few longer transition sequences. The step increase is a compromise between the depth-first and breadth-first and can check if a test target is unreachable under a given number of transitions.

The gain brought by parallel processing can be analyzed from the experiments conducted on the ServiDirect model. In the different strategies, there is a limit to the gain. In case of use of more than six threads, the gain becomes negligible or in the worst case the performance can be degraded. Indeed the probability to interrupt a thread depends on the number of threads and the interdependency of the targets.

Technically, a user does not need to have knowledge of the model to generate tests with the described process. However, without knowledge, the strategy and the parameters will be arbitrarily chosen. An improvement would be to have the capacity to classify the model to select the best strategy. The criteria can be the interdependency of the variables found in the class diagram or the complexity (number of outgoing and input UML4MBT transitions) of the state-chart.

From these results, three improvements can be considered in the future. A first would be to create new strategies by combining existing ones, for example, starting with the step-increase strategy to find the test targets with a limited number of transitions and then using the depth-first strategy to find targets that need longer transition sequences. A second one is to change the values of the parameters during the execution of a strategy. For instance, to reduce the number of animations and to limit the resolution time in the step strategy, the parameter ‘increase’, controlling the increase of the animation length, can be reduced during the execution of the strategy. A third improvement would be to select the order in which the test targets have to be reached, but this would require to add additional information in the model. For example, it can indicate that the target ‘isRepaired’ can be reached only after the target ‘isBroken’.

10. RELATED WORKS

As UML/OCL is currently the most employed modelling language to design systems, there is a real issue to verify the models or produce tests from a model. Whatever the initial focus, to verify the model or to generate tests, the methods go through a conversion of the model into a suitable formalism. To the best of our knowledge, no method takes into account the entire UML formalism; they are limited to the class diagram plus one or two other diagrams.

Thus, in the context of model checking process associated with MDD, J. Cabot, R. Clariso and D. Riera proposed to translate the class diagram as a Constraint Satisfaction Problem (CSP) [17]. They use a constraint solver to check the existence of a solution. The search for a solution by an enumeration process of the constraint variables, that is classical in CSP resolution, requires to define finite domains on each element of the model. In the approach presented in this paper, we have a similar restriction: all instances of the model have to be known at the initial state, and the integer attributes are bounded.

Also in the field of verification of UML/OCL, Mr. Soeken, R. Wille and R. Drechsler [18] proposed a translation of the successive states of the model in bit vector theory and then use an SMT prover to verify that each operation can be activated, that some system states are reachable or that the system is viable (it is not put in a deadlock situation). The process used in this work is very close to our work; it differs in the chosen system state encoding as well as the purpose. When they check the reachability of an operation, in order to validate the model, we generate a test case that can be used to validate the final application.

Similarly, M. Clavel, M. Egea and M. A. Garcia de Dios give a mapping of UML/OCL to first-order logic [19]. This allows to check different notions of satisfiability for the model. This process is close to ours. The major differences are that our encoding process does not use quantifiers and must take into account that sequences of multiple operations are considered.

K. Yatake and T. Aoki [20] describe the use of an SMT prover to exhaustively generate all instances of object diagrams from a UML model of the environment of an OS. Under considered conditions, namely a limited number of instances for the model classes, they calculate all test cases corresponding to the environments to apply to the operating system. This approach of test case generation is a bit different from ours. The model concerns the environment of the SUT rather than the behaviour of the SUT. So they do not produce any expected results for a test case.

Another approach is to use higher-order logic to represent the OCL constraints. This logic is described in HOL-OCL [21] and can be used by the theorem prover Isabelle. However, this prover is not automatic and requires interaction with a user.

Another way to generate tests from UML models, but different from ours, is to use symbolic automata, like in the tool AGATHA [22]. They transform UML models into an IOLTS formalism and then use symbolic execution to generate tests. This method has two phases: the creation of a symbolic execution tree and the valuation of a sequence of operations discovered with the tree by a constraint solver. Our method does not need to compute the execution tree. The evaluation is realized during the construction of the animation.

The performance of the SMT solvers has led others to use these tools with other contexts and languages. For example, A. Armando, J. Mantovani and L. Platania propose a tool [23], named SMT-CMBC, that uses a SMT solver to check the satisfiability of a logic formula, encoding a program trace of bounded length, that violates a property. These formulas use linear arithmetic, the theory of arrays and the theory of the bit vectors. Therefore, even if the encoding process shares some aspects like the Single State Assignment form, our method differs because we choose to use uninterpreted function to encode a collection. Moreover, the expressivity of UML4MBT and OCL4MBT is different.

Another example would be the work done by F. Büttner, M. Egea and J. Cabot [24] where the partial correctness of a declarative, rule-based transformation between constrained metamodels is checked with a SMT solver. The rules are defined in the ATL language, and the constraints are written in OCL. Their encoding process to obtain first-order logic formulas uses quantifiers and therefore can lead in some cases to inconclusive results because the solver cannot determine if a solution exists. Moreover, to differentiate between entities, we choose to use uninterpreted types where they use integer with specific constraints.

The Z3 SMT solver is also used by the SpecExplorer Tool [25], developed by Microsoft Research, that has been integrated into the Visual Studio IDE. This tool can generate tests from models. The behaviours of the model are represented through abstract state machines written in Spec# or AsmL. The test generation process creates an execution graph by exploration. The SMT solver is used to generate the values of the parameters of the operations during an execution.

CONCLUSION

In this paper, an end-to-end process to automatically generate tests from a UML4MBT model with an SMT solver has been presented. The method used is based on the translation of the class diagram, the object diagram and the state-chart diagram into first-order logic formulas. The subset of UML/OCL is defined with a metamodel. To obtain an efficient transformation between the two paradigms, an SMT metamodel has also been defined. This metamodel captures all required information to perform the animation of the model. The transformation rules are implemented from the metamodel definition, and the time resolution needed by the solver for the test generation is studied.

The time depends on the formula produced. To reduce the test generation time, five strategies of animation have been created and the process has been parallelized. The strategies use different approaches like the minimization of the length of the animation for the depth-first and breadth-first strategies, the minimization of the number of animations for the step increase strategy and the minimization of the number of constraints in the SMT instances for the path strategy. The efficiency of the process has been studied through experiments conducted on different real life systems. The experiments confirm the interest of the method, and interesting results have been obtained with the different strategies with regards to the model's characteristics.

In further work, we want to upgrade our test generation process in four ways. The first is to establish a classification of the models to select automatically the best strategies with the best parameters. The second is to add the possibility to describe a test target with the Business Process Modelling Notation because this language is becoming more and more important in the industry. The third is to create a new strategy by combining an SMT solver and a CSP prover. The CSP prover has the advantage of quickly generating different solutions and therefore can enumerate different states of the system compatible with a specific animation. The fourth is currently done. The process has been upgraded to be compatible with the new version of SMT-LIB.

APPENDIX A: OPERATIONS OF THE ROBOT MODEL

This part is dedicated to the description of the operations introduced by the class model of running example at Section 4. The operations are written in OCL4MBT and describe the dynamic of the Robot model.

A.1. Description of robot class operations

```
operation
  rotate(pos : HPosition)
pre
  (pos = HPosition::right
  and self.vPosition = VPosition::up
  and self.hPosition = HPosition::left
  and (self.item = Item::T2
    or ( self.item = Item::T3
      and self.upRightConveyorBelt.item = Item::NONE
      and not ( self.upLeftConveyorBelt.item = Item::NONE ))))
or  (pos = HPosition::left
  and self.vPosition = VPosition::up
  and self.hPosition = HPosition::right
  and (self.item = Item::NONE
    or (self.item = Item::T3
      and self.upLeftConveyorBelt.item = Item::NONE)))
post
  if
    pos = HPosition::right
  then
    self.hPosition = HPosition::right
  else
    self.hPosition = HPosition::left
  endif

operation
  translate(pos : VPosition)
pre
    (self.vPosition = VPosition::down
    and not (self.item = Item::NONE))
  or  (self.vPosition = VPosition::up
    and self.item = Item::NONE
    and self.hPosition = HPosition::left)
post
  if
    pos = VPosition::up
  then
    self.vPosition = VPosition::up
  else
    self.vPosition = VPosition::down
  endif
```

```
operation
  grab()
pre
  self.item = Item::NONE
  and not (self.downLeftConveyorBelt.item = Item::NONE)
  and self.hPosition = HPosition::left
  and self.vPosition = VPosition::down
post
  self.item = self.downLeftConveyorBelt.item
  and self.downLeftConveyorBelt.item = Item::NONE

operation
  drop()
pre
  self.vPosition = VPosition::up
  and ((self.upLeftConveyorBelt.item = Item::NONE
      and self.hPosition = HPosition::left
      and (self.item = Item::T1
        or  self.item = Item::T3))
    or (self.upRightConveyorBelt.piece = Item::NONE
      and self.hPosition = HPosition::right
      and (self.item = Item::T2
        or (self.item = Item::T3
          and not (self.upLeftConveyorBelt.item = Item::NONE)))))
post
  if
    self.hPosition = HPosition::left
  then
    self.upLeftConveyorBelt.item = self.item
    and self.item = Item::NONE
  else
    self.upRightConveyorBelt.item = self.item
    and self.item = Item::NONE
  endif
```

A.2. Description of incomingconveyorbelt class operations

```
operation
  nextItem(item : Item)
pre
  self.item = Item::NONE and item <> Item::NONE
post
  self.item = item
```

A.3. Description of outgoingconveyorbelt class operations

```
operation
  evacuate()
pre
  not (self.item = Item::NONE)
post
  self.item = Item::NONE
```

REFERENCES

1. Muller P-A, Fondement F, Baudry B, Combemale B. Modeling modeling modeling. *Software & Systems Modeling* 2012; **11**(3):347–359. DOI: 10.1007/s10270-010-0172-x. Available from: http://www.springerlink.com/content/f734260111w08jj2/ [last accessed 14 May 2014].
2. Barrett C, Deters M, Oliveras A, Stump A. Design and results of the 3rd Annual Satisfiability Modulo Theories Competition (SMT-COMP 2007). *International Journal on Artificial Intelligence Tools* 2008; **17**(4):569–606.
3. Ranise S, Tinelli C. The SMT-LIB standard: version 1.2, *Tech. Rep*, Department of Computer Science, The University of Iowa, 2006.
4. De Moura L, Bjørner N. Z3: An efficient SMT solver. *Tools and Algorithms for the Construction and Analysis of Systems* 2008; 337–340.
5. Damm W, Hermanns H, Barrett C, Tinelli C. *Computer aided verification*, Lecture Notes in Computer Science, vol. 4590. Springer Berlin Heidelberg: Berlin, Heidelberg, 2007. doi:10.1007/978-3-540-73368-3.

6. Barrett C, Stump A, Tinelli C. *Logics in The Satisfiability Modulo Theories Library (SMT-LIB)*, 2011. Available from: goedel.cs.uiowa.edu/smtlib/logics.html [last accessed 14 May 2014].
7. Halpern JY. Presburger arithmetic with unary predicates is Pi 1 1 complete. *Journal of Symbolic Logic* 1991; **56**: 56–2.
8. Dutertre B, De Moura L. *The Yices SMT solver*, 2006. Available from: *http://yices.csl.sri.com/tool-paper.pdf*. http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.85.7567&rep=rep1&type=pdf [last accessed 14 May 2014].
9. Bouquet F, Grandpierre C, Legeard B, Peureux F, Vacelet N, Utting M. A subset of precise UML for model-based testing. *Proceedings of the 3rd International Workshop on Advances in Model-Based Testing,* London, ACM, 2007; 104.
10. Steinberg D, Budinsky F, Paternostro M, Merks Ed. *EMF: eclipse modeling framework*, (2nd edn). Addison-Wesley: Boston, MA, 2009. Available from: http://my.safaribooksonline.com/9780321331885 [last accessed 14 May 2014].
11. *Deliverable 7.4: results of test campaign on case studies*, 2012. SecureChange (EU ICT-FET-231101).
12. Fraser G, Gargantini A. An evaluation of model checkers for specification based test case generation. *ICST*: IEEE Computer Society, 2009; 41–50.
13. Beizer B. *Software testing techniques*, 2nd ed. Van Nostrand Reinhold Co.: New York, NY, USA, 1990.
14. Cantenot J, Ambert F, Bouquet F. Transformation rules from UML4MBT meta-model to SMT meta-model for model animation. *12th Int. Workshop OCL, co-located with Int. Conf. MODELS 2012*, Innsbruck, Austria, 2012; 55–60.
15. Kröning D, Rümmer P. A proposal for a theory of Finite sets, lists, and maps for the SMT-LIB standard. *Informal Proceedings, 7th International Workshop on Satisfiability Modulo Theories at CADE 22*, Montreal, Canada, 2009. Available from: http://www.kroening.com/smt-lib-lsm.pdf [last accessed 14 May 2014].
16. Cytron R, Ferrante J. An efficient method of computing static single assignment form. *Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, Austin, Texas, 1989; 25–35.
17. Cabot J, Clarisó R, Riera D. Verification of UML/OCL class diagrams using constraint programming. *IEEE International Conference on Software Testing Verification and Validation Workshop, 2008. ICSTW'08,* Lillehammer, Norway, 2008; 73–80. Available from: http://scholar.google.fr/scholar?hl=fr&q=uml++sat+solver&btnG=Rechercher&lr=&as_ylo=&as_vis=0#9 [last accessed 14 May 2014].
18. Soeken M. Verifying dynamic aspects of UML models. *Design, Automation and Test in Europe Conference and Exhibition*, Grenoble, France, 2011; 1–6.
19. Clavel M, Egea M, García de Dios MA. Checking unsatisfiability for OCL constraints. *Electronic Communications of the EASST* 2010; **24**.
20. Yatake K, Aoki T. SMT-based enumeration of object graphs from UML class diagrams. *ACM SIGSOFT Software Engineering Notes* 2012; **37**(4):1–8. DOI: 10.1145/2237796.2237812. Available from: http://doi.acm.org/10.1145/2237796.2237812 [last accessed 14 May 2014].
21. Brucker AD, Wolff B. The HOL-OCL book. *Swiss Federal Institute of Technology (ETH)* 2006; **525**.
22. Bigot C, Faivre A, Gallois J-P, Lapitre A, Lugato D, Pierron J-Y, Rapin N. Automatic test generation with AGATHA. In *Tacas*, vol. 2619, Garavel H, Hatcliff J (eds), Lecture Notes in Computer Science. Springer: Heidelberg, 2003; 591–596.
23. Armando A, Mantovani J, Platania L. Bounded model checking of software using SMT solvers instead of SAT solvers. *International Journal on Software Tools for Technology Transfer* 2009; **11**(1):69–83.
24. Büttner F, Egea M, Cabot J. On verifying ATL transformations using 'off-the-shelf'SMT solvers. In *Model Driven Engineering Languages and Systems*, France RB *et al.* (eds). Springer: Berlin Heidelberg, 2012; 432–448.
25. Veanes M, Campbell C, Grieskamp W, Schulte W, Tillmann N, Nachmanson L. Model-based testing of object-oriented reactive systems with Spec Explorer. In *Formal Methods and Testing*, Hierons RM *et al.* (eds). Springer: Berlin Heildelberg, 2008; 39–76.