Transformation rules from UML4MBT meta-model to SMT meta-model for model animation

Jérôme Cantenot
University of Franche-Comté
Femto-ST institute – France
jcantenot@femto-st.fr

Fabrice Ambert
University of Franche-Comté
Femto-ST institute – France
fambert@femto-st.fr

Fabrice Bouquet
University of Franche-Comté
Femto-ST institute – France
fbouquet@femto-st.fr

ABSTRACT

In the Model Based Testing domain, one of the bottleneck elements is the tests generation (time or reachability). In literature, one of the efficient solutions to evaluate a formula is to use an SMT solver. However the input language, SMT-lib, used by the solvers is not adapted to human modelling. To model we choose to use a sufficient sub-part of UML/OCL for Model Based Testing generation, called UML4MBT. This sub-part has been formalized with a meta-model. In this paper, we define the SMT meta-model and we describe the transformation rules from the UML4MBT meta-model to the SMT meta-model. We give the results of the experiments on a first implementation of the rules.

Categories and Subject Descriptors

D.2.1 [**Requirement and Specification**]: Languages, Tools

; D.2.4 [**Software Verification**]: Validation

; D.2.5 [**Testing**]: Symbolic execution

General Terms

Verification, Experimentation

Introduction

A tool needs to fulfil two main requirements to be useful in the context of model based testing. The first one is to have a good expressiveness because we have to design precise models of complex real system. The second one is to have high performance because the size and complexity of models make it difficult to generate tests quickly. We choose to use Satisfiability Modulo Theories solvers because these tools meet the requirements. There is a common input language for SMT solvers named SMT-lib. However this language has little-to-no semantics and is not adapted to model a system. On the contrary, the UML4MBT language, a subset of UML, has good semantics and has been created for the model based testing. Our goal is to translate automatically a UML4MBT model into an SMT model. The SMT model can then be submitted to an SMT solver to generate tests. In this paper, we focus on the model translation. The other steps of the process will be treated in further articles.

The idea to represent a model written in UML/OCL with a set of constraints using First Order Logic is not new. Works to convert a model into other languages like B [10], SAT predicates [5] or Alloy [1] exist. The main difference between SMT formulas and Alloy or B is the purpose of the language. B and Alloy are high-level languages that have a common goal: to model a system. For example, B has operations and Alloy has parametrized predicates to represent an action on the model. On the contrary, SMT model uses a low-level language that does not possess mechanisms to allow the animation of the model. Moreover the SMT model does not support set theory. We have difference on previous works [6],[13] by having a different encoding and taking into account other diagrams.

The remainder of the paper is organized as follows. Section 1 describes the UML4MBT meta-model, constructed from the UML4MBT language. This language is a subset of UML created for the model based testing, described in the paper [4]. Section 2 presents the SMT meta-model constructed from the SMT-lib language [2]. This language provides a common input language for the majority of SMT solvers. Sections 3,4 and 5 list a set of rules to process a conversion between a UML4MBT meta-model and an SMT meta-model. Section 6 presents an example to validate our conversion rules.

1. UML4MBT MODEL

The UML4MBT language is a subset of the UML notation, described in the paper [4], created specifically for the Model-Based Testing. UML4MBT uses the UML 2.0 language as a modeling notation for tests generation. This means that a specific point of view on the system will be adopted and that only class, state and object diagrams of the UML language will be used.

The class diagram describes the structure of the system and its behaviour. This diagram shows classes, attributes, operations and the relationships between classes. The state diagram describes the behaviour of the system. This diagram is optional because we can choose to model the behaviour only with operations. The object diagram shows a view of the system in an initial state.

UML4MBT has some restrictions. The inheritance relationship is not authorized. Another restriction is that all instances must be known explicitly. Therefore the object diagram must contain all instances used during the execution of the system. This restriction is acceptable because when we test a system in real condition, we cannot have an infinite number of instances.

To express constraints on the elements of the system, we use the Object Constraint Language in conjunction with the UML4MBT language. In practice, we use a specific version named OCL4MBT constructed from the OCL language and created for the Model-Based Testing. The OCL4MBT language uses two separate contexts. The first, named "evaluation context", respects the standard behaviour of OCL. This context is used during the evaluation of a condition which can be the pre-condition of an operation, the guard of a transition or the condition in the "if then else" operator. In the second context, named "assignment context", OCL4MBT is an imperative language. OCL4MBT contradicts standard semantics and therefore UML4MBT with OCL4MBT cannot be a UML profile.

The meta-model has been realized by the MIPS team (`http://www.mips.uha.fr`). Principles used to construct this meta-model are described in the paper [11]. This meta-model is constructed with the Eclipse Modeling Framework. A particularity of this meta-model is that there are no string or real types and integer are bounded because we want to have the capacity to enumerate all potential values.

2. SMT MODEL

A Satisfiability Modulo Theories solver can determine if a First-Order Logic (FOL) formula, where some function and predicate symbols have additional interpretations, is satisfiable. There is a common language to communicate with solvers and to describe background theories named the Satisfiability Modulo Theories Library. This allows us to choose from a wide range of solvers like Z3 [8], CVC3 [3] or YICES. The version 1.2 of the language, described in the paper [12], fills most of our needs except for sets. Among SMT-lib's logics, we choose to use 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 reals signature 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.

We choose to use this logic for the following reasons. First, the basic types found in the UML4MBT meta-model can be encoded with linear real arithmetic. Then current state-of-the-art SMT solvers cannot always conclude when formulas are not quantifier free. Indeed, there is no sound and complete procedure for FOL formulas of linear arithmetic with uninterpreted function symbols [9]. And that is why we do not use quantifiers. Finally we need uninterpreted functions to encode set theory.

The SMT-lib has little-to-no semantics. An SMT model must contains enough information to represent a system and to be submitted to an SMT solver. The SMT meta-model contains the minimum entities to respect this definition. Figure 1 shows the class diagram of the SMT meta-model. A benchmark is the major entity that contains a formula, a list of assumptions and the necessary information to process this formula. When we use an SMT solver, we need to specify a logic. We can also indicate the satisfiability of the formula to increase the performance. Indeed, the SMT solvers can use different optimization techniques with this information. A formula or an assumption is a predicate, written in the QF-UFLRA logic. The difference is that an assumption is implicitly true. A benchmark is therefore satisfiable if and only if the formula are true under the assumptions. Another entity is an SMT function. An SMT function can represent a variable or a mathematical function. In this paper, function means SMT function. A function can have one or many types. The types are restricted to primitive types (Integer, Named and Boolean) with the addition of the set type. The named type allows to use uninterpreted functions.

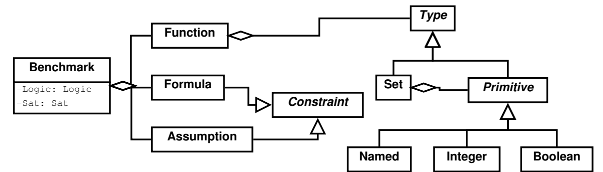

Figure 1: Class diagram of the SMT meta-model

The constraints can use all operators defined in the QF-UFLRA logic and two basic set operators: "cardinality" and "membership". We have the membership operator because in this model, sets are represented by a membership relation. Moreover we can iterate over any set, because all potential set elements are known. Therefore we can easily deduce the cardinality of a set.

3. RULES CONCERNING UML4MBT

In this section, we will present the rules to convert static parts of a UML4MBT model described by the object and the class diagrams.

For these rules, a model can be represented by a 4-Tuple (Class, Slot, Enum, Link). **Class** is a set of model classes. We introduce the notation "Inst(A)" meaning "all instances of class A". **Slot** is a set of slots belonging to model instances. **Enum** is a set of model enumerations. We introduce the notation "Enum(A)" meaning "all literals in the enumeration A". **Link** is a set of model links. The links represents the associations between instances.

Table 1 contains conversion rules. The left part of the table contains conditions on the UML4MBT entities to trigger the conversion rule. In conditions, x represents the entity to convert. The right part describes the effect of conversion rules. A rule can create SMT sorts, corresponding to types in the SMT meta-model, SMT functions and SMT constraints.

To explicit how a conversion rule works, we detail the conversion of an enumeration named "fruit" containing literals "apple" and "pear". First we select rules that have a satisfied condition. In this case, we select the rule named "Enum" because the condition $\mathit{fruit} \in \mathit{Enum}$ is true. The right part of the table indicates that we must create: a sort named "fruit", 2 functions named "apple" and "pear" of "fruit" type and a constraint "distinct apple pear".

Classes are converted into SMT sorts and instances are converted into SMT functions. Moreover, we must specify that each instance is a different object through an SMT constraint. To represent the null value, we create a special function named x_{none}.

Slots are converted into SMT functions of the correct sorts. In the case of range and enumeration types, we need SMT constraints to bound the value of the created functions.

The QF-UFLRA logic cannot express natively constraints on the set theory. In our case, we use sets to represent associations between classes or collections created from existing sets of instances. To overcome this problem, we convert the set A that contains all potential instances of class C into an SMT function $A : C \to \{0, 1\}$. Thus the function A associates 1 if the instance is actually present in the set, and 0 otherwise. These functions can be understood as membership functions. In the special case where the cardinality of the set is equal to one, we can replace this set by an SMT function. In this case, we obtain a rule similar to the instance of a class.

Rules about *association* are valid only in the case of unidirectional association. But we can replace a bidirectional association by two unidirectional associations and some constraints. We can model the multiplicity of an association by two constraints on the cardinality of set. These two rules are named “Link-set” and “Link-single” in the table 1.

Table 1: Rules for UML4MBT model

	UML4MBT	SMT		
Name	Condition	Sort	Function	Constraint
Slot-int	$x \in Slot$ $x \in \mathbb{N}$		$x : Int$	
Slot-bool	$x \in Slot$ $x \in \{false, true\}$		$x : Bool$	
Slot-range	$x \in Slot$, $x \in \mathbb{N}$ $x_{min} \leq x \leq x_{max}$		$x : Int$	$x \leq x_{max}$ $x \geq x_{min}$
Slot-enum	$x \in Slot$ $x \in Enum$		$x : enum$	$\exists e \in enum : x = e$
Class	$x \in Class$ $Inst(x) = \{i_0, i_1, \ldots, i_n\}$	x	$x_{none} : x$	$distinct\ i_0, \ldots, i_n$
Instance	$x \in Inst(class)$		$x : class$	
Enum	$x \in Enum$	x	$\forall e \in Enum(x) : e : x$	$distinct\ e_0, \ldots, e_n$
Link-single	$x = x_{a \to b} \in Link$ $max(\lvert x \rvert) = 1$ $Inst(b) = \{i_0, i_1, \ldots, i_n\}$		$x : b$	$\bigvee_{j=0}^{n} x = i_j$
Link-set	$x = x_{a \to b} \in Link$ $max(\lvert x \rvert) > 1$ $Inst(b) = \{i_0, i_1, \ldots, i_n\}$		$x : b \to Int$	$\bigwedge_{j=0}^{n} x(i_j) = 0 \vee x(i_j) = 1$

4. RULES CONCERNING OCL4MBT

In this section and the following, rules are described in a table. The left part of the table contains a condition on the OCL4MBT entities to trigger the conversion rule. The right part have 3 columns named Formula, Function and Constraint. The Function and Constraint columns list the SMT functions and constraints to add to the SMT model. The Formula column contains the current translation of the OCL4MBT formula. Moreover the conversion process is recursive; if we can, we execute a rule on the formula described in the Formula column.

Table 2 contains the rules that are common to both the assignment and the evaluation contexts. For these rules, we have $a, b \in Class^2$, $Inst(b) = \{b_0, b_1, \ldots, b_n\}$, var is a variable representing an instance of b and $slot_{bi}$ 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.

Table 2: Rules for OCL4MBT without context

	UML	SMT		
Rule name	Condition	Formula	Function	Constraint
Int-Access	$var.slot$ $slot \in \mathbb{N}$	$access$	$access : Int$	$\bigwedge_{i=0}^{n} var = b_i \to access = slot_{bi}$
Bool-Access	$var.slot$ $slot \in \{false, true\}$	$access$	$access : Bool$	$\bigwedge_{i=0}^{n} var = b_i \to access = slot_{bi}$
Enum-Access	$var.slot$ $slot \in Enum$	$access$	$access : enum$	$\bigwedge_{i=0}^{n} var = b_i \to access = slot_{bi}$
Link-Access	$var.slot$ $slot = slot_{a \to b} \in Link$	$access$	$access : b$	$\bigwedge_{i=0}^{n} var = b_i \to access = slot_{bi}$

OCL4MBT is a contextual language. Therefore the meanings of operators depends on the context. In the first subsection, we detail the conversion of operators in an assignment context where OCL4MBT is an imperative language. In the second subsection, we present the conversion of operators in an evaluation context where the language respects the semantic of the OCL language.

4.1 Rules in assignment context

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 in the SMT language, we choose to use static single assignment form that is defined in [7] to assure the uniqueness of each assigned functions . And that is why we must create a new SMT function for each assignment.

During the conversion process, a UML4MBT entity can be converted into multiple SMT functions. To create a link between all these functions, we introduce 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 3 contains rules to convert OCL4MBT code in the assignment context. For all rules in this table, we have $i_a \in Inst(c)$, $Inst(c) = \{i_0, i_1, \ldots i_n\}$ and $c \in Class$. We will only highlight a few facts. When we assign a null value to an instance in the rule “A-undefined”, we use a custom value created specifically to represent this case. As we said above, when we work with sets, we cannot use quantifiers. And that is why we need to iterate over the elements of the set. In this case, we obtain a more complex but sound formula.

The conversion of an operation call is more complex. An operation call is composed by 3 parts: an operation with a pre/post definition, 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 4 steps. The first is to determine if the call is valid. If the predicate representing the ”pre” definition of the operation can be satisfiable 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” definition 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 third step is to create a constraint that implies the ”post” definition of

Table 3: Rules in assignment context

	OCL4MBT	SMT	
Name	Formula	Formula	Function
A-int	$in_1, in_2 \in \mathbb{N}^2$ $in_1 = in_2$	$\varphi(in_1) = in_2$	$\varphi(in_1) : Int$
A-bool	$b_1, b_2 \in \{false, true\}^2$ $b_1 = b_2$	$\varphi(b_1) = b_2$	$\varphi(b_1) : Bool$
A-enum	$l_1 \in Slot$ $l_1, l_2 \in Enum(e_3)$ $e_3 \in Enum$ $l_1 = l_2$	$\varphi(l_1) = l_2$	$\varphi(l_1) : Enum$
A-inst	$i_1 = i_2$	$\varphi(i_1) = i_2$	$\varphi(i_1) : class$
A-undef	$i_a.oclIsUndefined()$	$\varphi(i_a) = class_{none}$	$\varphi(i_a) : class$
A-set	$S_1 = S_2$ $S_1, S_2 \subseteq Inst(c)^2$	$\bigwedge_{i=0}^{n} \varphi(S_1(i_j)) = S_2(i_i)$	$\varphi(S_1) : c \to Int$
A-include	$S_1 \subseteq Inst(c)$ $S_1.includes(i_a)$	$\bigwedge_{j=0}^{a-1} \varphi(S_1(i_j)) = S_1(i_j)$ $\varphi(S_1(i_a)) = 1$ $\bigwedge_{j=a+1}^{n} \varphi(S_1(i_j)) = S_1(i_j)$	$\varphi(S_1) : c \to Int$
A-exclude	$S_1 \subseteq Inst(c)$ $S_1.excludes(i_a)$	$\bigwedge_{j=0}^{a-1} \varphi(S_1(i_j)) = S_1(i_j)$ $\varphi(S_1(i_a)) = 0$ $\bigwedge_{j=a+1}^{n} \varphi(S_1(i_j)) = S_1(i_j)$	$\varphi(S_1) : c \to Int$
A-empty	$S_1.empty()$ $S_1 \subseteq Inst(c)$	$\bigwedge_{j=0}^{n} \varphi(S_1(i_j)) = 0$	$\varphi(S_1) : c \to Int$
A-forAll	$S1.forAll(expr(i_j))$ $S_1 \subseteq Inst(c)$ $expr : c \to Bool$	$\bigwedge_{j=0}^{n} expr(i_j)$	

the operation if the boolean variable is true. The last step is to substitute the input parameters by their values.

4.2 Rules in evaluation context

Table 4: Rules for boolean operators in evaluation

	UML	SMT
Rule name	Condition	Formula
E-bool-op2	$b_1 \diamondsuit b_2$	$b_1 \diamondsuit b_2$
E-bool-ite	$if\ b_1\ then\ b_2\ else\ b_3$	$ite\ b_1\ \phi(b_2)\ \phi(b_3)$
E-bool-false (true)	$false\ (true)$	$false\ (true)$
E-bool-slot	$b_1 \in Slot$	$\varphi(b_1)$

This section presents rules to convert OCL4MBT code under the evaluation context. Table 4 contains rules for boolean operators. For all rules, we have $b_1, b_2, b_3 \in \{false, true\}^3$. These operators have a direct equivalent and have a straight conversion. $\diamondsuit$ represents any binary operator $(=, \neq, \vee, \wedge, \oplus)$. The only difficulty concerns the rule "E-bool-ite" because we use single state assignment form. If we consider the formula $if\ b\ then\ b = false\ else\ true$, a naive conversion will give $ite\, b_0\ (b_1 = false)\ true$. However in this case, we obtain a different result to $\varphi(b)$ for each branch. In the "then" branch, we have $\varphi(b) = b_1$. In the "else" branch, we have $\varphi(b) = b_0$. We use a standard method in single static assignment form, the ϕ function, to solve this problem. This method is described in paper [7]. The ϕ function allows to obtain the same result for each branch. In our example, we get $ite\, b_0\ (b_1 = false)\ ((b_1 = b_0) \wedge\ true)$.

Table 5: Rules for integer operators in evaluation

	UML	SMT		
Rule name	Condition	Formula	Function	Constraint
E-int-op2	$i_1 \diamondsuit i_2$	$i_1 \diamondsuit i_2$		
E-int-minus-1	$-i_1$	$-i_1$		
E-int-div	$i_1.div(i_2)$	$div\ i_1\ i_2$		
E-int-abs	$i_1.abs()$	i_2	$i_2 : Int$	$ite\ (i_1 < 0)\ (i_2 = (-i_1))\ (i_2 = i_1)$
E-int-max	$i_1.max(i_2)$	i_3	$i_3 : Int$	$ite\ (i_1 < i_2)\ (i_3 = i_2)\ (i_3 = i_1)$
E-int-min	$i_1.min(i_2)$	i_3	$i_3 : Int$	$ite\ (i_1 > i_2)\ (i_3 = i_2)\ (i_3 = i_1)$
E-int-number	$i_1 \notin Slot$	i_1		
E-int-slot	$i_1 \in Slot$	$\varphi(i_1)$		

Table 5 contains rules for the integer operators. In this table, we have $i_1, i_2, i_3 \in \mathbb{N}^3$. $\diamondsuit$ represents any binary operator $(=, \neq, <, >, \leqslant, \geq, +, -, *)$. There is direct equivalence except for rules "E-int-abs", "E-int-max" and "E-int-min". These rules create a new function to store the result of a test.

Table 6: Rules for enum in evaluation

	UML	SMT
Rule name	Condition	Formula
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	$l_1 \notin Slot$	l_1
E-enum-slot	$l_1 \in Slot$	$\varphi(l_1)$

Table 6 contains rules for the enumeration operators. For all rules in this table, we have $e \in Enum$, $l_1, l_2 \in Enum(e)^2$.

Table 7: Rules for instances and class in evaluation

	UML	SMT		
Rule name	Condition	Formula	Function	Constraint
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	$\varphi(i_1)$		
E-inst-allInstance	$c.allInstance()$	c_{all}	$c_{all} : c \to Int$	$\bigwedge_{j=0}^{n} c_{all}(i_j) = 1$

Table 7 contains rules for instance and class operators. For all rules in this table, we have $c \in Class$, $\{i_0, i_1, \ldots, i_n\} = Inst(c)$. The rule "E-inst-undefined" gives the null value to a function. We recall that the null value is represented by a specific function created during the conversion of the class of the model. However with this mechanism, all null values are different. Each class has a null value that can be seen like a particular instance. The rule "E-inst-allInstance" returns a function that represent a set containing all instances of a class. This conversion uses the fact that we know all existing instances at the start of the animation.

Table 8: Rules for set operators in evaluation

	UML	SMT		
Rule name	Condition	Formula	Function	Constraint
ES-equal	$S_1 = S_2$	$\bigwedge_{j=0}^{n} S_1(i_j) = S_2(i_j)$		
ES-inequal	$S_1 \neq S_2$	$\neg(\bigwedge_{j=0}^{n} S_1(i_j) = S_2(i_j))$		
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	$S_1.includeAll(S_2)$	$\bigwedge_{j=0}^{n} (S_2(i_j) = 1)$ $\to (S_1(i_j) = 1)$		
ES-exclAll	$S_1.excludeAll(S_2)$	$\bigwedge_{j=0}^{n} (S_2(i_j) = 1)$ $\to (S_1(i_j) = 0)$		
ES-empty	$S_1.isEmpty()$	$\sum_{j=0}^{n} S_1(i_j) = 0$		
ES-notEmp	$S_1.notEmpty()$	$\neg(\sum_{j=0}^{n} S_1(i_j) = 0)$		
ES-union	$S_1.union(S2)$	S_3	$S_3 : c \to Int$	$\bigwedge_{j=0}^{n} ite\ (S_2(i_j) + S_1(i_j) = 0)$ $(S_3(i_j) = 0)\ (S_3(i_j) = 1)$
ES-inter	$S_1.inter(S2)$	S_3	$S_3 : c \to Int$	$\bigwedge_{j=0}^{n} ite\ (S_2(i_j) + S_1(i_j) = 2)$ $(S_3(i_j) = 1)\ (S_3(i_j) = 0)$
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}	$i_{any} : c$	$ite\ expr(i_0)\ (i_{any} = i_0)$ $(ite\ expr(i_1)\ (i_{any} = i_1)$ $(\ldots ite\ expr(i_n)\ (i_{any} = i_n)$ $(i_{any} = i_{none}))\ldots)$
ES-value	S_1	$\varphi(S_1)$		

Table 8 contains rules for the set operators. For all rules in this table, we have $c \in Class$, $S_1, S_2, S_3 \subseteq Inst(c)^3$, $Inst(c) = \{i_0, i_1, \ldots, i_n\}$ and $expr : c \to Int$. We recall that sets are modelled by a memberships functions that return 1 if the element is present in the set and 0 otherwise. This property allows to write the rules "ES-include" and "ES-exclude". With this model, the cardinality of a set is equivalent to the sum of the function applied to all elements of the set (see rule "ES-size"). Rule "ES-equal" defines that two sets are equal if they contains the same elements. Rule "ES-empty" uses the fact that a set is empty if the cardinality of the set is 0. Rules "ES-union" and "ES-inter" start by creating a function representing a new set and then add a constraint to put correct elements in this set.

Under the evaluation context, the OCL4MBT has the restriction that an operation call must not modify the state of the system. The conversion has 4 parts. Like in the assignment context, the "pre" definition 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" definition of the operation called must be created.

5. RULES FOR THE ANIMATION

We first establish some definitions. A state of the model is defined by a valuation of all or some variables in the model. To modify the state of the model, we must execute one operation or transition. This execution is called a step. An animation is a sequence of steps.

To animate a model, its transition diagram must follow these two restrictions: do not have parallel states and transitions are triggered only by operations. And furthermore, the number of animation steps must be fixed at the beginning of the animation. This constraint is due to the interaction with the solver.

For the reasons discussed in the section about assignment, for each step, we must duplicate SMT functions that do not represent a constant. Therefore, we must also duplicate constraints using these functions.

We introduce the following notations: **State** and **Trs** list respectively the states and transitions of the model, **Ops** list operations found in the model, **pre(x)** and **post(x)** returns respectively the pre/post-condition of the operation x or the guard/action of the transition x, **in(x)** and **out(x)** returns the source (resp. the target) state of the transition(x) and **var(x)** returns all SMT functions that represent a variable.

Table 9 contains rules to animate the model. For all rules in this table, we have $\{a_0, a_1, \ldots, a_n\} = var(x)$ and the word *next* indicates a function used by the next step. The rule **"State"**, creates an SMT function to represent each state found in the UML state-chart;

The rule **"Operation"** creates three functions for each operation found in the class diagram. The function x represents the operation, $trig_x$ is a flag to memorize if this operation has been activated during the last step and $step_x$ is a boolean function that is true if this operation is executed during this step. The constraint created by this rule can be read as: if the precondition of the operation x is verified and this operation is animated during this step, then we memorized for the next step its activation, we execute the postcondition and we assign new values to state variables, else we memorized that this operation has not been chosen.

The rule **"Transition"** follows the same principle. The constraint is modified because a transition can be executed only if the model is in a correct state and the trigger has been activated (see $trig(x) \wedge in(x) = state$ in the formula). In the same way, if the transition is activated, we need to memorize the new state of the system and to reset the trigger function.

The rule **"Model"** has two goals. The first is to create a lock function named "event". This function with the first two constraints forces the solver to execute one and only one transition or operation during an animation step. The second goal is to create a dummy operation named op_{none}. Without this operation, we can obtain an unsatisfiable formula when we reach a final state of state-chart and we have not terminated the animation.

Table 9: Rules for the model animation

	UML	SMT		
Name	Condition	Sort	Function	Constraint
State	$x \in State$		$x : State$	
Operation	$x \in Ops$		$x : Event$ $trig_x : Bool$ $step_x : Bool$	$ite\ (x = event \wedge pre(x))$ $(trig_{x_{next}} = true \wedge post(x) \wedge$ $\bigwedge_{i=0}^{n} a_{i_{next}} = \varphi(a_i))$ $(step_x = false)$
Transition	$x \in Trs$		$x : Event$ $step_x : Bool$	$ite\ (x = event \wedge pre(x) \wedge trig(x)$ $\wedge in(x) = state)$ $(trig_{none_{next}} = true$ $\wedge \bigwedge_{i=0}^{n} a_{i_{next}} = \varphi(a_i)$ $\wedge state_{next} = out(x))$ $(step_x = false)$
Model		$Event$ $State$	$event : Event$ $op_{none} : Event$ $trig_{none} : Bool$ $state : State$ $step_{none} : Bool$	$distinct\ tr_0, \ldots, tr_n, op_0, \ldots, op_p$ $\bigvee_{i=0}^{n} step_{tr_i} \vee \bigvee_{i=0}^{p} step_{op_i} \vee step_{none}$ $ite\ (x = op_{none})$ $(trig_{none_{next}} = true$ $\wedge \bigwedge_{i=0}^{n} a_{i_{next}} = a_i)$ $(step_{none} = false)$

6. CONVERSION SOUNDNESS

In this section, we present two kinds of example conversion to confirm the soundness of the approach. These examples are added to unit test realized for each rule. The first example covers all rules to validate our conversion process on a simple model. In the second example, we test the scalability of our process on a more complex model. All animations are realized with a specific SMT solver: Z3 (v2.19).

6.1 Coverage

In this section, we will validate all conversion rules on a simple example. We have modelled a PID (process id) system. This system uses a scheduler to manage a pool of threads. Figure 2 shows the class and the object diagram. The scheduler can activate, delete, create or swap a thread. The state diagram of thread life cycle has three states: waiting, ready and active.

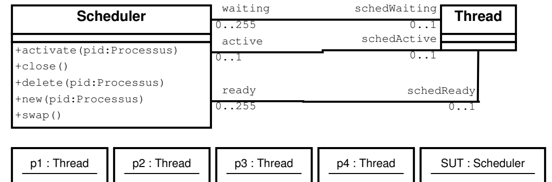

Figure 2: Class & object diagram of the PID system

We want to verify if we can activate one transition or operation from the initial state. In this case, an animation of one step is sufficient. The first stage is to convert the static part of the model. We begin with the rule, named "Class" presented in table 1. After the conversion, we have two sorts named "Thread" and "Scheduler". Then we use the rule "Instance" to create functions: $p1$ to $p4 : Thread$ and $SUT : Scheduler$. We will convert links on demand.

The second stage is to convert the dynamic part of the model. Table 10 shows the conversion process of the guard and action, written in OCL4MBT, of the transition named waiting2ready. The last line of each part of the table is written in SMT metamodel. We highlight that we use assignment rules only during the conversion of the action. Others

transitions and operations are converted in the same way. At the end of conversion, we get an SMT model containing 4 sorts, 137 functions and 89 constraints. This SMT model represents the evolution of the system after one transition of the state-chart or operation. This model can then be written in the smt-lib language and submitted to an SMT solver. Z3 found a solution to this problem in 0.01 seconds. We have proved that an operation or transition can be activated from the initial state of the model. Of course, the size of the SMT model increases linearly with the number of steps in the animation. The conversion of this model used all rules described in this paper.

Table 10: Conversion of waiting2ready

Guard	$self.active.oclIsUndefined()\ and\ self.waiting.includes(pid)$
ES-value:	$active_0.oclIsUndefined()\ and\ waiting_0.include(pid)$
E-inst-value:	$active_0.oclIsUndefined()\ and\ waiting_0.include(pid_0)$
E-inst-undefined:	$active_0 = Thread_{none}\ and\ waiting_0.include(pid_0)$
ES-include:	$active_0 = Thread_{none}\ and\ if\ waiting_0(pid_0) = 1\ then\ true\ else\ false$
E-bool-and:	$active_0 = Thread_{none}\ and\ if\ waiting_0(pid_0) = 1\ then\ true\ else\ false$
Action	$self.waiting.excludes(pid)\ and\ self.active = pid$
ES-value:	$waiting_0.excludes(pid)\ and\ active_1 = pid$
E-inst-value:	$waiting_0.excludes(pid_0)\ and\ active_1 = pid_0$
A-inst:	$waiting_0.excludes(pid_0)\ and\ active_1 = pid_0$
A-exclude:	$waiting_1\ and\ active_1 = pid_0$ (add new assumption on $waiting_1$)
E-bool-and:	$waiting_1\ and\ active_1 = pid_0$

6.2 Scalability

To validate some choices about rules, we use a second model realized during the European project Test_Indus (`http://disc.univ-fcomte.fr/test_indus`), that represents a real website (`https://www.servidirect.com`) selling medical insurances named "ServiDirect". In this model, we have more interactions between instances. The model contains a person associated with his children, his residence, his insurance policy. The model contains 9 classes and 26 instances.

Our process makes two choices that can have a serious impact on the scalability. The first choice is to use the single static assignment method [7]. This method increases the number of SMT functions and constraints when the number of steps in the animation increases. We study the resolution time in function of the number of steps in the animation. The model contains $621 * steps + 888$ SMT functions and $1329 * steps + 868$ constraints. With this model, we can simulate a continuous sequence of 5,10,15 and 20 operations and transitions in respectively 9, 35, 110 and 464 seconds.

The second choice is to remove all quantifiers in formulas to help Z3 to find a solution. We study the evolution of the resolution time in function of the number of instances. Each instance is associated with at least another instance through an association. All animations have a size of 10 steps. In the model with 12, 52 and 102 instances, we have respectively 4933, 7913 and 11638 SMT functions and the solver found a solution in respectively 6, 91 and 352 seconds.

Conclusion

In this paper, we have defined the conversion rules to create an SMT model from a UML4MBT model. We also have described a method to rewrite set operators in the first order logic when all potential elements of sets are known. This automatic transformation allows to animate the resulting SMT model in order to simulate the behaviour of the system. So the SMT solvers can determine if a state of the model is reachable. The fact that we can reach a dedicated state in the model permits to generate tests in a Model Based Testing approach. Finally we have validated the conversion process and its scalability on two models.

The next step is to create new strategies for the tests generation. Indeed, the SMT model contains constraints representing the static, dynamic parts and of the initial state of the system. The initial state is the starting point of the search space. If we isolate these constraints, we change the starting point of the search space and therefore create more efficient strategies. Another approach is to parallelize the research. Moreover the SMT language is independent of solvers, so we can test different solvers with the same model.

This work has been supported by the city of Besançon and the Franche-Comté region.

7. REFERENCES

[1] K. Anastasakis, B. Bordbar, G. Georg, and I. Ray. *Model Driven Engineering Languages and Systems*, volume 4735 of *LNCS*. Springer, Berlin, 2007.

[2] C. Barrett, A. Stump, and C. Tinelli. The SMT-LIB Standard Version 2.0. 2010.

[3] C. Barrett and C. Tinelli. Cvc3. In W. Damm and H. Hermanns, editors, *Computer Aided Verification*, volume 4590 of *LNCS*, pages 298–302. Springer, 2007.

[4] F. Bouquet, C. Grandpierre, B. Legeard, F. Peureux, N. Vacelet, and M. Utting. A subset of precise uml for model-based testing. In *Proceedings of the 3rd Int. WS*, A-MOST '07, pages 95–104, London, 2007. ACM.

[5] J. Cabot, R. Clarisó, and D. Riera. Verification of UML/OCL class diagrams using constraint programming. In *IEEE Int. Conf. on Software Testing Verification and Validation Workshop, 2008. ICSTW'08*, pages 73–80, 2008.

[6] M. Clavel, M. Egea, and M. A. G. de Dios. Checking unsatisfiability for ocl constraints. *ECEASST*, 24, 2009.

[7] R. Cytron and J. Ferrante. An efficient method of computing static single assignment form. *Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, 1989.

[8] L. De Moura and N. Bjørner. Z3: An efficient SMT solver. *Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340, 2008.

[9] J. Y. Halpern. Presburger arithmetic with unary predicates is pi 1 1 complete. *Journal of Symbolic Logic*, 56:56–2, 1991.

[10] H. Ledang and J. Souquières. Integration of UML and B specification techniques: systematic transformation from OCL expressions into B. In *Software Engineering Conference*, pages 495–504, 2002.

[11] P.-A. Muller, F. Fondement, B. Baudry, and B. Combemale. Modeling modeling modeling. *Software & Systems Modeling*, pages 1–13–13, Aug. 2010.

[12] S. Ranise and C. Tinelli. The smt-lib standard: Version 1.2. *Department of Computer Science, The University of Iowa, Tech. Rep*, 2006.

[13] K. Yatake and T. Aoki. Smt-based enumeration of object graphs from uml class diagrams. *SIGSOFT Softw. Eng. Notes*, 37(4):1–8, July 2012.