Verification of UML/OCL Class Diagrams using Constraint Programming

Jordi Cabot, Robert Clarisó and Daniel Riera
Universitat Oberta de Catalunya
{jcabot, rclariso, drierat}@uoc.edu

Abstract

In the MDD and MDA approaches, models become the primary artifacts of the development process. Therefore, assessment of the correctness of such models is a key issue to ensure the quality of the final application. In that sense, this paper presents an automatic method that uses the Constraint Programming paradigm to verify UML class diagrams extended with OCL constraints. In our approach, both class diagrams and OCL constraints are translated into a Constraint Satisfaction Problem. Then, compliance of the diagram with respect to several correctness properties such as weak and strong satisfiability or absence of constraint redundancies can be formally verified.

1 Introduction

Software verification is one of the long-standing goals of software engineering. The need for correct software specifications is even more relevant in the context of the MDD and MDA communities where software models are used to (semi)automatically generate the implementation of the final software system.

Unfortunately, formal verification of software models is known to be undecidable in general. This is also the case when focusing on the verification of UML class diagrams extended with OCL constraints: first-order logic (FOL) itself is undecidable in general and OCL is more expressive than FOL. Therefore, to avoid undecidability, existing methods able to reason on UML/OCL diagrams either (a) limit the UML/OCL constructs that may appear in the diagrams, (b) are not automatic or (c) are semi-decidable.

We believe that these limitations impair a wide adoption of formal methods within the MDD community. As a consequence, specification and design errors are not detected until the implementation stage, increasing the cost of the development process.

In this paper we advocate for using the Constraint Programming paradigm as a complementary method for the fully automatic, decidable and expressive verification of UML/OCL class diagrams. Decidability is achieved by defining a *finite* solution space, i.e. establishing finite bounds for the number of instances and finite domains for attribute values to be considered during the verification process. This way, the constraint solver is able to perform a *complete search* within the solution space. We will argue that considering a finite solution space is a reasonable trade-off regarding the features offered by other existing verification methods.

The main goal of this paper is to present a systematic procedure for the transformation of a UML class diagram annotated with OCL constraints into a Constraint Satisfaction Problem (CSP). A predefined set of correctness properties about the original UML/OCL diagram can then be checked on the resulting CSP.

One of the most well-known correctness properties is *satisfiability*. A model is satisfiable if it is possible to create a correct and non-empty instantiation of the model, i.e. if a user can possibly create a finite set of new objects and links over the classes and associations of the model so that no model constraint is violated. As an example, consider the class diagram of Fig. 1. This model is unsatisfiable due to two different reasons:

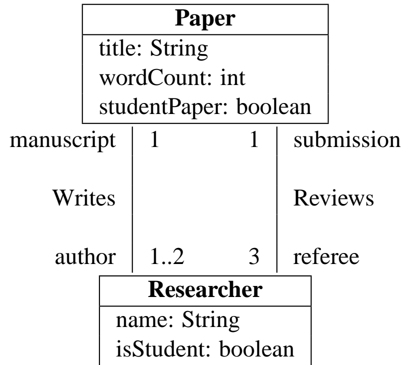
1. The multiplicities of association *Reviews* require exactly three distinct researchers per paper ($|Researcher| = 3 \cdot |Paper|$). Meanwhile, the multiplicities of *Writes* requires one or two researchers per paper ($|Researcher| \leq 2 \cdot |Paper|$). Only an infinite or empty instantiation may satisfy both constraints simultaneously.

2. Students cannot be referees according to constraint *NoStudentReviewers*. However, all researchers must be authors (due to the multiplicities in *Writes*), all authors must review papers (*Reviews*) and there must be at least one student paper (*LimitsOnStudentPapers*) with an student author (*AuthorsOfStudentPaper*).

Therefore, the model we have presented is completely useless[1]. Every time a user tries to instantiate the model some of the constraints will become violated.

Roughly, to detect the unsatisfiability of this model our method would proceed as follows. The diagram is translated into a CSP, such that if the CSP has a solution, the model is satisfiable. Intuitively, the CSP describes an instance of the model using variables that encode the number

[1] Fig.1 becomes satisfiable if the multiplicities of $manuscript$ and $submission$ are changed to $0..1$. This version of the model will be used later in the paper to illustrate a successful search.




```
context Researcher inv NoSelfReviews:
    self.submission−>excludes(self.manuscript)

context Paper inv PaperLength:
    self.wordCount < 10000

context Paper inv AuthorsOfStudentPaper:
    self.studentPaper = self.author−>exists(x | x.isStudent )

context Paper inv NoStudentReviewers:
    self.referee−>forAll(r | not r.isStudent)

context Paper inv LimitsOnStudentPapers:
    Paper::allInstances()−>exists(p | p.studentPaper) and
    Paper::allInstances()−>select(p | p.studentPaper) −>size() < 5
```

Figure 1. Running example: a UML class diagram with OCL constraints

of objects and links, the values of attributes, etc. In the CSP, there are also several constraints that restrict the legal values for these variables and that represent, for instance, OCL constraints or multiplicities of the initial model. Satisfiability (non-emptiness of the instantiation) can be imposed as an additional constraint: a lower bound on the number of objects and links.

To find a solution, the constraint solver tries to assign a value to all variables without violating any constraint. If no legal assignment is possible, the model is determined as unsatisfiable. Likewise, other correctness properties can be checked.

The rest of the paper is structured as follows. Section 2 introduces Constraint Programming concepts and notation. Later, Section 3 describes how to transform a UML/OCL model into a CSP. Section 4 presents some correctness properties and their representation as additional constraints in the CSP. The resolution of the generated CSP is shown in section 5. The verification tool that implements our approach is introduced in Section 6. Previous work and theoretical aspects are analysed in Section 7. Finally, Section 8 draws some conclusions and highlights future work.

2 Basic concepts of Constraint Programming

Constraint Programming [2, 10] is a declarative problem solving paradigm where the programming process is limited to the definition of the set of requirements (constraints). A *constraint solver* is in charge of finding a solution that satisfies the requirements.

Problems addressed by Constraint Programming are called *constraint satisfaction problems* (CSPs). A CSP is represented by the tuple CSP = $\langle V, D, C\rangle$ where V denotes the finite set of *variables* of the CSP, D the set of *domains*, one for each variable, and C the set of *constraints* over the variables. A *solution* to a CSP is an assignment of values to variables that satisfies all constraints, with each value within the domain of the corresponding variable. A CSP that does not have solutions is called *unfeasible*.

The most traditional technique for finding solutions to a CSP is backtracking. A possible backtracking implementation called *labeling* orders variables according to some heuristic and attempts to assign values to variables in that order. If any constraint is violated by a partial solution, the solver reconsiders the last assignment, trying a new value in the domain and backtracking to previous variables if there are no more values. This systematic search continues until a solution is found or all possible assignments have been considered. To ensure termination, the search space must be finite, thus, all variable domains must be finite.

The efficiency of the search process is largely improved by *constraint propagation* techniques: using information about the structure of constraints and the decisions taken so far in the search process, the unfeasible values in the domains of unassigned variables can be identified, pruning the search tree. Theses techniques are an effective mechanism to reduce the search space.

In this paper, we will describe CSPs using the syntax provided by the ECL^{i}PS^e Constraint Programming System [2,15]. In ECL^{i}PS^e, constraints are expressed as predicates in a logic Prolog-based language while variables may be either *simple*, *structured* (tuples) or *lists*. The environment provides several solvers and it is capable of reasoning about boolean, interval, linear and arithmetic constraints among others.

3 Translation of UML/OCL Class Diagrams

This section describes the transformation of a class diagram into a Constraint Satisfaction Problem. A class diagram *CD* is defined as *CD* = $\langle Cl, As, AC, G, IC\rangle$, where Cl is the set of classes, As is the set of associations, AC the set of association classes, G the set of generalisation sets and IC the set of constraints (either graphical or textual) included in *CD*.

Each element is translated into a set of variables, domains and constraints in the CSP system. As stated before, domains must be finite. These finite domains can be ensured in several ways: first of all, arbitrary bounds for the domains can be chosen or provided by the designer during the translation process. On the other hand, the analysis of the constraints in IC may reveal a finite set of relevant values in the domain. From the point of view of efficiency, we are interested in the smallest domains that suffice to identify inconsistencies in the model, but the automatic computation of these domains from the constraints in IC is a complex problem which will not be addressed in this paper. Instead, we will assume in this section that these values are provided as inputs (parameters) of our translation procedure.

In the following we present the transformation of the elements of a class diagram into the CSP.

3.1 Transformation of classes

The set of variables and domains to be defined for each class $c \in Cl$ is:

- A variable $Instances_c$ of type list. Each element in the list represents an instance of c. Therefore, the domain of these elements is represented by the structure $struct(c) = (oid, f_1, \ldots, f_n)$, where: oid represents the explicit object identifier for each object, and each f_i corresponds to an attribute $at \in$ *c.ownedAttribute*[2].

  The domain of the oid field is the set of positive integers. The domain of an f_i field is defined as a finite subset of the domain of the corresponding at attribute in c. Boolean and enumerated types are already finite. Finite domains for integer types requires at least a lower and upper bound for the attribute. For real types we need also a maximum decimal precision. For string types, the possible "alphabet" and the maximum string length should be defined.

  To increase the efficiency of the generated CSP, during the translation we discard all attributes that do not participate in any of the constraints in IC. A correct instantiation may contain any value in those attributes.

- A variable $Size_c$ of type integer, encoding the number of instances of class c. Its domain is $domain(Size_c) = [0, PMaxSize_c]$, where $PMaxSize_c$ is a parameter that indicates the maximum number of instances of class c that must be considered when looking for a solution to the CSP.

Additionally, the following constraints are added to the CSP:

- *Number of instances*: $Size_c = \text{length}(Instances_c)$
- *Distinct oids*: $\forall x, y \in Instances_c : x \neq y \rightarrow x.oid \neq y.oid$

3.2 Transformation of associations

For each association $as \in As$ between classes $C_1 \ldots C_n$, the following variables and domains must be created in the CSP:

- A variable $Instances_{as}$ of type list. Every member of the list represents an instance of the association (i.e. a link), each being of type $struct(as) = (p_1, \ldots, p_n)$, where $p_1 \ldots p_n$ are the role names of the participant classes. The domain of each p_i is that of positive integers, that is, each link records the set of oids of the participant objects, not the objects themselves.

- A variable $Size_{as}$ encoding the number of instances of the association. Its domain is $domain(Size_{as}) = [0, PMaxSize_{as}]$. As before, $PMaxSize_{as}$ is the parameter indicating the maximum number of links of as to be considered when looking for valid solutions of the CSP.

Let n be the number of roles in the association as, and given a role i, let $T(i)$ be its type and $[m_i, M_i]$ be its multiplicity. Then, the following constraints must also be added to the CSP:

- *Number of links*: $Size_{as} = \text{length}(Instances_{as})$
- *Existence of referenced objects*: $\forall l \in Instances_{as} : \forall i \in [1, n] : \exists x \in Instances_{T(i)} : x.oid = l.p_i$
- *Uniqueness of links*: $\forall x, y \in Instances_{as} : x \neq y \rightarrow (\exists i \in [1, n] : x.p_i \neq y.p_i)$ unless the property *isUnique* of the association is set to false.
- *Bounds on cardinalities*: The multiplicities of an association impose constraints on the number of instances of the participant classes and the association. These constraints are presented in Fig. 2. First, the set of links is a subset of the cartesian product of the participant classes, so its size (product of class sizes) defines an upper bound for the number of links. Also, minimum and maximum multiplicities of roles define a lower and upper bound relationship between the number of links and the number of objects of each participant class.
- *Multiplicities of the association*: Multiplicity constraints must also be satisfied by each individual object of the participant classes. For instance, for a binary association, the condition

[2] *ownedAttribute* is the UML metamodel navigation that returns the set of attributes of a class.

Class X	$m_a..M_a$	Assoc A	$m_b..M_b$	Class Y
	$role_a$		$role_b$	

$$Size_A \leq Size_X \cdot Size_Y$$
$$m_a \cdot Size_Y \leq Size_A \leq M_a \cdot Size_Y$$
$$m_b \cdot Size_X \leq Size_A \leq M_b \cdot Size_X$$

Figure 2. Implicit cardinality constraints due to the association multiplicities [5]

$$(\forall x \in Instances_{T(1)} : m_2 \leq \{\#l : l \in Instances_{as} : l.p_1 = x\} \leq M_2) \ \wedge$$
$$(\forall y \in Instances_{T(2)} : m_1 \leq \{\#l : l \in Instances_{as} : l.p_2 = y\} \leq M_1).$$

is the constraint imposed by min/max multiplicities.

3.3 Transformation of association classes

An association class $ac \in Ac$ is, at the same time, a class and an association. Therefore, transformation of association classes can be regarded as the union of the translation process for classes plus the translation process for associations.

3.4 Transformation of generalisation sets

Generalisation sets do not imply the definition of new variables but the addition of new constraints among the classes involved in the generalisation set.

Let class $sub \in Cl$ be a subclass of a class $super \in Cl$. The following constraints should be added:

- *Existence of oids in supertype*: $\forall x \in Instances_{sub} : \exists y \in Instances_{super} : x.oid = y.oid$
- *Number of instances*: $Size_{sub} \leq Size_{sup}$
- *Disjointness*: For a disjoint generalization set among a supertype S and subtypes $S_1..S_n$:
  - $Size_S \geq \sum_i Size_{Si}$
  - $\forall i, j \in [1, n] : \forall o_1 \in Instances_{Si}, \forall o_2 \in Instances_{Sj} : o_1.oid = o_2.oid \rightarrow i = j$
- *Completeness*: For a complete generalization set among a supertype S and subtypes $S_1..S_n$:
  - $Size_S \leq \sum_i Size_{Si}$
  - $\forall o_1 \in Instances_S : \exists i \in [1, n] : \exists o_2 \in Instances_{Si} : o_1.oid = o_2.oid$

3.5 Translation of OCL invariants

Integrity constraints in OCL [11] are represented as invariants defined in the context of a specific type, named the *context type* of the constraint. Its body, the boolean condition to be checked, must be satisfied by all instances of the context type. In our approach, each OCL constraint is translated into an equivalent constraint in the CSP. Fig. 3 shows an example of the translation process presented in this section. Note that the same translation process could be seamlessly used to translate other OCL expressions like pre and postconditions.

An OCL constraint can be viewed as an instance of the OCL metamodel with a tree shape (see the simplified tree representation for *PaperLength* constraint in Fig. 3). Leave nodes of the tree correspond to the constants (e.g. 2, *true*, "John") and variables (e.g. *self*, x) of the constraint. Each internal node corresponds to one atomic operation of the constraint, e.g. logical or arithmetic operation, access to an attribute, operation calls, iterator, etc. The root of the tree is the most external operation of the constraint. Packages like the Dresden OCL toolkit [6] can parse textual OCL constraints and build the corresponding trees.

As a preliminary step, we express all constraints in terms of the *allInstances* operation using the following expansion rule:

context T **inv**: B $\Rightarrow$

context T **inv**: T::allInstances()$-$>forAll(v|B')

where B' is obtained by replacing all occurrences of *self* in B with v.

Then, the translation procedure is defined as a post-order traversal of the corresponding OCL metamodel tree that translates all the children (subexpressions) of a node before translating the node (expression) itself. Each node of the tree is translated into an ECL^{i}PS^e Prolog compound term with an *unique functor name* that identifies the subexpression and *three arguments*, e.g. `nodeX(Instances, Vars, Result)`, with the following meaning:

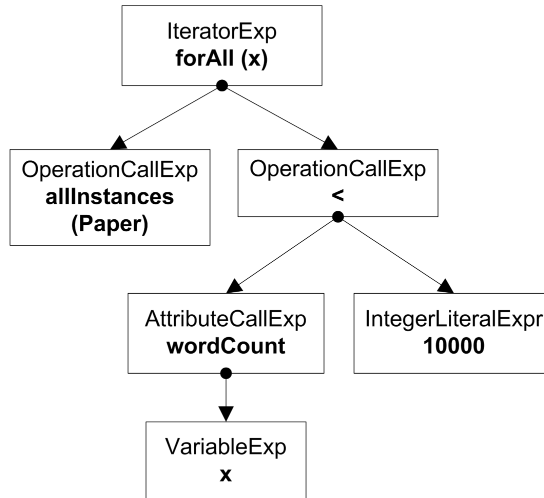
1. *Instances* is a list with the set of *instances* for each class and association. The i-th position of this list holds all the instances of class/association i. The order within this list is defined in auxiliary Prolog rules generated during the translation. This argument is required, for instance, to implement the OCL operation *allInstances* and navigation in associations.

2. *Vars* contains the list of the *quantified variables* available in the subexpression. The first position of this list holds the value of the quantified variable defined in the innermost iterator (e.g. *forAll* or *exists*). The second position holds the following variable in the next innermost iterator and so on. This argument will be used when evaluating attribute, operation or navigation expressions over variables defined in an iterator.

3. *Result* holds the *result* of the subexpression. The type of the result depends on the operation applied in the node.

context Paper **inv** PaperLength:
Paper::allInstances−>
forAll(x|x.wordCount < 10000)

(a)



(b)

```
% Position of class Paper
% within the list of instances
index("Paper", 1).

% Position of attribute wordCount
% within the list of attributes
attIndex("Paper", "wordCount", 2).

nodeConstant(_, _, Result):-
  Result = 10000.

nodeVariable(_, Vars, Result ):-            % x = var of the innermost iterator
   nth1(1, Vars, Result).                   % Result = Vars[1] = value of x

nodeAttrib(Instances, Vars, Result):-
   nodeVariable(Instances, Vars, Object),   % An object of class Paper
   attIndex("Paper", "wordCount", N),       % N = Index of field wordCount
   arg(N, Object, Result).                  % Result = Object[N] = wordCount value

nodeAllInstances(Instances, Vars, Result) :-
   index("Paper", N),                       % N = Position of class Paper
   nth1(N, Instances, Result).              % Result = Instances[N] = Inst of Paper

nodeLessThan(Instances, Vars, Result) :-
   nodeAttrib(Instances, Vars, Value1),     % 1st subexpression
   nodeConstant(Instances, Vars, Value2),   % 2nd subexpression
   #<(Value1, Value2, Result).              % Result = (Value1 < Value2)?

nodeForAll(Instances, Vars, Result) :-
   nodeAllInstances(Instances, Vars, L),    % L = Result of allInstances
   ( foreach(Elem, L), foreach(Eval, Out), param(Instances,Vars) do
       % Eval = Result of evaluating nodeLessThan on an element of L
       nodeLessThan(Instances, [Elem|Vars], Eval) ),
   % Out = List of truth values.  Out[i]= Result of nodeLessThan(L[i])
   length(L, N),                            % N = length(L)
   #=(N, sum(Out), Result).                 % Result = (N = ΣOut[i])?

% Translation of the constraint PaperLength
paperLength(Instances) :-
   nodeForAll(Instances,[],Result),         % Evaluate the root node
   Result #= 1.                             % Result should be true
```

(c)

Figure 3. Translation of OCL constraints: (a) Class invariant after preprocessing, (b) OCL metamodel tree, (c) Constraint represented by means of Prolog rules in the CSP

The behaviour of each node is formalised by means of a Prolog *rule*. This rule evaluates the subexpressions of the node and computes the result of the node in terms of the results of its subexpressions. Basic types (e.g. boolean, integer or real) and basic operations (e.g. logical and arithmetic) have a direct implementation in the ECLiPSe constraint libraries. For more complex operations, such as iterators or operations on Collections, we have developed a new ECLiPSe library [16] that implements the operations defined in the OCL Standard Library [11]. This library is implemented such that embedded constraint propagation techniques in ECLiPSe can be applied. Nevertheless, for the sake of simplicity, in Fig. 3 we have directly added to each node the required computation without relying in our external library.

Once the translation has been completed, we add to the CSP a new constraint representing the original OCL invariant, defined as: `nameConstraint(Instances):-rootNode(Instances,[],Result),Result#=1`, i.e. a constraint is true when `rootNode` evaluates to true. For example, see the *paperLength* constraint in Fig. 3(c).

4 Definition of correctness properties

A model is expected to satisfy several reasonable assumptions. For instance, it should be possible to instantiate the model in some way that does not violate any integrity constraint. Moreover, it may be desirable to avoid unnecessary constraints in the model. Failing to satisfy these criteria may be a symptom of an incomplete, over-constrained or incorrect model. Designers can select which of these criteria should be satisfied by a model.

In our approach, correctness properties are represented as additional constraints in the CSP. If the CSP still has a solution once the new constraint is added, we may conclude that the model satisfies the property. The set of correctness properties that can be checked by designers is the following:

Strong satisfiability: The model must have a finite instantiation where the population of all classes and associations is at least one.

Weak satisfiability: The model must have a finite instantiation where the population of at least one class is at least one.

Liveliness of a class c**:** The model must have a finite instantiation where the population of c is non-empty.

Lack of constraint subsumptions: Given two integrity constraints C_1 and C_2, the model must have a finite instantiation where C_1 is satisfied and C_2 is not. Otherwise, we say that C_1 *subsumes* C_2. C_2 could be removed.

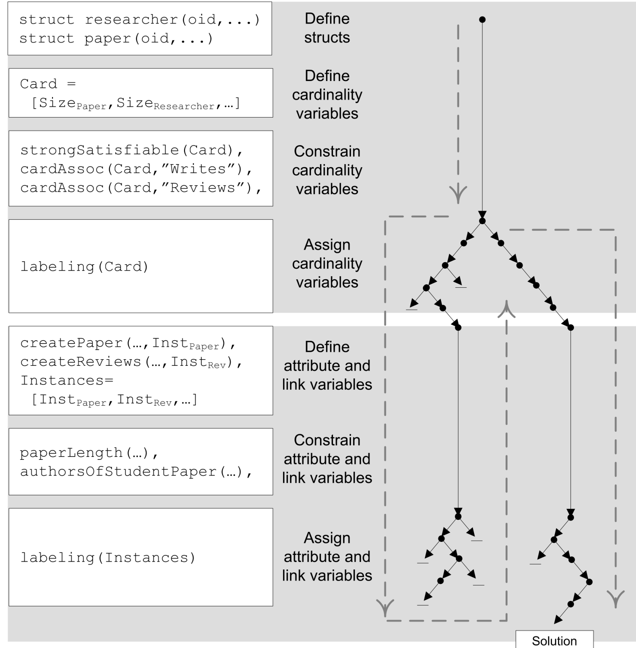


Figure 4. Definition of the CSP for the running example

Lack of constraint redundancies: Given two integrity constraints C_1 and C_2, the model must have a finite instantiation where only one constraint is satisfied. Otherwise, constraints C_1 and C_2 are called *redundant*, e.g. both have always the same truth value. One of them should be removed.

Other types of correctness properties, such as the *applicability of an operation op* (that is, verifying the existence of at least a valid instantiation where the precondition of *op* is satisfied), may be similarly defined.

Designers may be also interested in checking these properties over specific (partially defined) instantiations, e.g. checking satisfiability when a class c has an instance with a value v in an attribute a. Our approach allows the definition of additional constraints that characterise this desired state.

5 Resolution of the generated CSP

The final CSP is obtained as a combination of the translation excerpts generated using the rules of section 3 (for the transformation of the UML/OCL diagram) and section 4 (for the definition of the quality properties to be verified). Remember that if this generated CSP has a solution, we can determine that the model satisfies the indicated quality properties.

The CSP is organized in two subproblems. In the first one, we define the cardinality variables for the number of instances of each class and association (the $Size_x$ variables),

their domains and all constraints restricting them. In this phase, the goal is to find a legal assignment of values to these variables [5]. If no assignment is possible, the CSP is directly unfeasible.

In the second subproblem, the valid values assigned to the $Size_x$ variables are used to instantiate the corresponding $Instances_x$ variables. Now the goal is to find legal values for properties (either attributes or roles) of all elements in the $Instances_x$ lists. Intuitively, the procedure tries to find a valid solution for this second subproblem for each assignment satisfying the first one. If there is no such solution, the CSP is determined as unfeasible.

Both phases follow the typical Constraint Programming outline: define the variables and their domains, define the constraints on the variables, and finally, find a legal assignment to these variables. In the initial phase, we work on cardinality variables ($Size_x$), while in the second phase we are interested in the set of instances ($Instances_x$) of classes and associations.

As an example, Fig. 4 depicts the CSP corresponding to a satisfiable version of our running example. The colored areas highlight the two subproblems of the CSP. On the left of the figure, the organisation of several code excerpts (some of them taken from previous figures) is described. On the right, a possible search tree is depicted, where a dotted line shows the direction of the search. In this tree, after an initial attempt, a solution to the first subproblem is found, but it is not possible to complete the second subproblem using those values as cardinalities for the $Instances_x$ variables. Therefore, it is necessary to find another solution to the first subproblem, which can then be completed to find a valid solution to the CSP.

6 Tool implementation

Our prototype tool [16] is implemented as a set of ECL^{i}PS^e constraint libraries (2000 LoC) and Java classes (11500 LoC) extended with the libraries of the Dresden OCL toolkit [6] (for the parsing and loading of OCL constraints) and MDR (for the import/export of UML models from XMI). This prototype addresses the verification of UML class diagrams with OCL invariants, i.e. the static component of OCL. Solutions to the CSP are displayed graphically as an object diagram which satisfies all constraints.

7 Related work

Typically, approaches devoted to the verification of UML/OCL class diagrams (as our own approach) transform the diagram into a formalism where efficient solvers or theorem provers are available. However, there are complexity and decidability issues to be considered. Reasoning on UML class diagrams is EXPTIME-complete [3] and, when general OCL constraints are allowed, it becomes undecidable. By choosing a particular formalism, each method commits to a different trade-off regarding the verification of correctness properties of UML/OCL diagrams.

Table 1 briefly compares the tool described in this paper, UMLtoCSP, to other related tools. For each approach, the following information is listed: the underlying formalism, the translation procedure from UML/OCL to the formalism (manual or automated), the degree of automation in the verification (user-assisted or automated) and other limitations of the method. UMLtoCSP offers both automated translation and verification procedures and supporting general OCL constraints. Additionally, our tool is able to provide valid instantiations for satisfiable models.

Among all these tools, the most similar in terms of features is the combination of two tools, Alloy [8] and UML2Alloy [1]. Alloy is a mature tool for the automated analysis of software specifications with a consolidated implementation, but its input notation has differences with respect to UML/OCL. A separate front-end called UML2Alloy [1] can transform UML class diagrams annotated with OCL constraints into the Alloy notation, for a specific subset of UML constructs and OCL expressions.

UMLtoCSP offers some advantages with respect to the combination of UML2Alloy and Alloy. First, Alloy works by transforming the entire problem into an instance of SAT (satisfiability of a boolean formula in conjunctive normal form). Numerical constraints must also be expressed in terms of boolean variables, meaning that arithmetic and relational operations (e.g. addition, difference, less-than, ...) must be encoded as boolean formulas operating at the bit-level. All these factors lead to a combinatorial explosion in the size of the formula when the bit-width of integers increases. Moreover, it is not possible to encode constraints involving multiplications or divisions, and floating point values are not allowed. In a CSP, increasing the range of a numeric value also increases the search space, but encoding complex arithmetic expressions on integers or floats is straightforward. Finally, another benefit of UMLtoCSP is a minor advantage in terms of usability, as UML2Alloy and Alloy are separate tools. Meanwhile, UMLtoCSP offers an integrated environment for verification, providing results completely automatically in a notation (an object diagram) which is directly linked to the original UML model.

Even though our current tool implementation does not support yet all the features in the OCL 2.0 specification (e.g. constraints on strings), our approach does not impose theoretical limitations that restrict any UML or OCL constructs, unlike other approaches. On the other hand, like all bounded verification methods, our approach is decidable but *not complete*: results are only conclusive if a solution to the CSP is found. In that sense, our method only guarantees that if a solution to the CSP exists within the parameters

Table 1. Comparison of several methods for the verification of UML/OCL class diagrams.

Tool	Formalism	Translation	Verification	Limitations
[3, 14]	Description Logics	Automatic	Automatic	No OCL support
[5, 9]	CSP	Manual	Automatic	No OCL support, bounded verification
Alloy [8]	Relational Logics	Manual or [1]	Automatic	Bounded verification, limited arithmetic support
HOL-OCL [4]	Higher-Order Logics	Automatic	User-assisted	Undecidability
CQC [12]	Deductive DB queries	Manual	Automatic	No support for OCL arithmetic expressions, non-termination for infinitely satisfiable models
USE [7]	ASSL	Manual	Automatic	Validation only
UMLtoCSP	CSP	Automatic	Automatic	Bounded verification

provided by the user, it will be discovered. Nevertheless, the absence of solutions within a finite search space cannot be used as a proof: a solution may still exist outside the search space defined by the parameters.

Nonetheless, an efficient decidable procedure may provide more useful information than a semidecidable procedure, even if the answer is not conclusive. For example, when checking for satisfiability, the maximum population value for classes and associations can be always kept low. In practice, it may be as problematic to have a non-satisfiable model as to have a model that to be satisfiable requires populating the classes with too many instances, e.g. a model that requires creating more than fifty instances of each class to be satisfiable may be unusable in practice and may deserve further inspection anyway.

8 Conclusions and Further Work

We have presented a fully automatic, decidable and expressive method for the formal verification of UML/OCL class diagrams. Our method is based on the translation of the class diagram into a CSP. This approach has been implemented in a prototype tool [16].

As a trade-off the verification procedure is not complete: the user must provide a set of parameters to limit the search space. Our procedure guarantees that this search space will be explored exhaustively. We believe this is a reasonable trade-off given the advantages of our method.

As a further work we would like to refine our translation process to improve the efficiency of the obtained CSP. In particular we plan to better study heuristics to guide the search and advance in the automatic definition of appropriate ranges for attribute domains (based on the semantics of the OCL constraints that reference them). We plan to validate these improvements by means of applying our method over industrial case studies, including domain-specific languages [13]. Finally, we plan to integrate into our method the verification of other UML diagrams.

References

[1] K. Anastasakis, B. Bordbar, G. Georg, and I. Ray. Uml2alloy: A challenging model transformation. In *ACM/IEEE 10th International Conference on Model Driven Engineering Languages and Systems*, pages 436–450, 2007.

[2] K. R. Apt and M. G. Wallace. *Constraint Logic Programming using* ECL^{i}PS^e. Cambridge University Press, Cambridge, UK, 2007.

[3] D. Berardi, D. Calvanese, and G. D. Giacomo. Reasoning on UML class diagrams. *Artificial Intelligence*, 168:70–118, 2005.

[4] A. D. Brucker and B. Wolff. The HOL-OCL book. Technical Report 525, ETH Zurich, 2006.

[5] M. Cadoli, D. Calvanese, G. D. Giacomo, and T. Mancini. Finite satisfiability of UML class diagrams by Constraint Programming. In *Proc. Int. Workshop on Description Logics (DL'2004)*, volume 104 of *CEUR Workshop Proc.*, 2004.

[6] B. Demuth. The Dresden OCL toolkit and its role in Information Systems development. In *Proc. of the 13th International Conference on Information Systems Development (ISD'2004)*, Vilnius, Lithuania, 2004.

[7] M. Gogolla, J. Bohling, and M. Richters. Validating UML and OCL models in USE by automatic snapshot generation. *Journal on Software and System Modeling*, 4(4):386–398, 2005.

[8] D. Jackson. Alloy: a lightweight object modelling notation. *ACM Transactions on Software Engineering and Methodology*, 11(2):256–290, 2002.

[9] H. Malgouyres and G. Motet. A UML model consistency verification approach based on meta-modeling formalization. In *Proc. ACM Symp. on Applied Computing (SAC'2006)*, pages 1804–1809. ACM Press, 2006.

[10] K. Marriott and P. J. Stuckey. *Programming with Constraints: An Introduction*. MIT Press, Cambridge, Massachussetts, 1998.

[11] Object Management Group. *UML 2.0 OCL Specification*, 2003.

[12] A. Queralt and E. Teniente. Reasoning on UML class diagrams with OCL constraints. In D. W. Embley, A. Olivé, and S. Ram, editors, *ER*, volume 4215 of *Lecture Notes in Computer Science*, pages 497–512. Springer-Verlag, 2006.

[13] S. Sen, B. Baudry, and H. Vangheluwe. Domain-specific model editors with model completion. In *Proc. of the Multi-Paradigm Modeling Workshop (MPM'2007)*, 2007.

[14] R. V. D. Straeten, T. Mens, J. Simmonds, and V. Jonckers. Using description logic to maintain consistency between UML models. In *Proc. of UML'03.*, volume 2863 of *LNCS*, pages 326–340. Springer, 2003.

[15] The ECL^{i}PS^e Constraint Programming System. `http://www.eclipse-clp.org`, mar 2007. version 5.10.

[16] UMLtoCSP. `http://gres.uoc.edu/UMLtoCSP`.