On the Verification of UML/OCL Class Diagrams using Constraint Programming

J. Cabot[a], R. Clarisó[b,*], D. Riera[b]

[a] *AtlanMod Research Group, INRIA/École des Mines de Nantes, 4 rue Alfred Kastler, F-44307 NANTES Cedex 3*
[b] *Estudis d'Informàtica, Multimèdia i Telecomunicació, Universitat Oberta de Catalunya, Rambla del Poblenou 156, 08018 Barcelona*

Abstract

Assessment of the correctness of software models is a key issue to ensure the quality of the final application. To this end, this paper presents an automatic method for the verification of UML class diagrams extended with OCL constraints. Our method checks compliance of the diagram with respect to several correctness properties including weak and strong satisfiability or absence of constraint redundancies among others. The method works by translating the UML/OCL model into a Constraint Satisfaction Problem (CSP) that is evaluated using state-of-the-art constraint solvers to determine the correctness of the initial model. Our approach is particularly relevant to current MDA and MDD methods where software models are the primary artifacts of the development process and the basis for the (semi-)automatic code-generation of the final application.

Keywords: UML, OCL, MDD, Model Verification, Constraint Programming, Constraint Satisfaction Problem

1. Introduction

One of the initial stages in the development of a software system is the definition of its *conceptual schema* [Oli07]. A conceptual schema is a description of the knowledge that will be used by the software system, which may include a collection of relevant concepts, attributes, relationships among concepts, integrity constraints or inference rules. There are many candidate notations for expressing a conceptual schema, either with a visual or textual syntax and with or without a formal semantics. To name a few, some of the most well-known notations are *entity-relationship (ER) diagrams* in database design [Cha76], *UML class diagrams* in software engineering [Obj11] and *logic-based*

*Corresponding author: Robert Clarisó, Estudis d'Informàtica, Multimèdia i Telecomunicació, Universitat Oberta de Catalunya, Rambla del Poblenou 156, 08018 Barcelona, Spain. phone: (+34)933263410 fax:(+34)933568822 e-mail: rclariso@uoc.edu

representations, *frames*, *semantic networks* and *conceptual graphs* in knowledge engineering [Sow00].

1.1. Consistency checking of UML/OCL models

From the point of view of software quality, some desirable properties of a conceptual schema are *completeness* (no relevant information is missing) and *consistency* (the schema does not contain contradictory information)[1]. Ensuring completeness requires some degree of expert validation. In contrast, consistency checking can be automated if the schema is described in a notation providing a formal semantics.

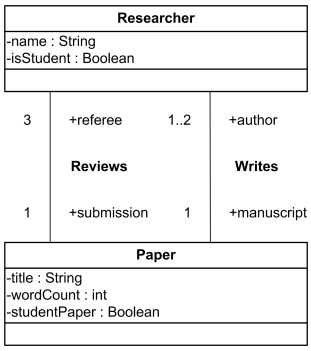
In the context of software engineering, conceptual schemas are typically described using ER diagrams or UML class diagrams. These notations provide several mechanisms to define relationships among concepts, such as associations or inheritance, and to describe integrity constraints, for instance, cardinality constraints. Even though these are simple constructs, the interactions among them may cause consistency problems which are hard to detect for human designers. Furthermore, UML diagrams can be annotated with more complex constraints written in the Object Constraint Language (OCL) [Obj10]. The addition of OCL invariants makes consistency problems even less intuitive, making the case for providing tool support to assist designers.

In addition to the *likelihood* of inconsistencies due to the expressivity of UML/OCL, there are two pragmatic reasons that motivate the study of the UML/OCL consistency problem, both related to development costs and software quality:

- First, it is an opportunity to detect errors early in the development process. Boehm's first law [ER03] states that "errors are most frequent during the requirements and design activities and are the more expensive the later they are removed". Thus, correcting consistency errors at this stage can help reduce development costs and even if the UML/OCL design is only being used for documentation purposes, checking consistency may be worthwhile.

- Second, in paradigms such as Model-Driven Development (MDD), inconsistencies may propagate directly into implementation errors. MDD advocates for considering models as the primary artifact of the development process. That is, after creating a model, it is used to (semi-)automatically generate the implementation of the final software system. In the case of UML/OCL class diagrams, code generation could translate the diagram into an object-oriented class hierarchy or database schema and transform the invariants into assertions, integrity constraints or declaratives queries

[1] In the literature, the terms consistency, correctness and satisfiability are sometimes used with different meanings. In this paper, we will use the terms *consistency* and *correctness* interchangeably, and we will assign a formal meaning to the term *satisfiability*.



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.

[CT06, HWD08]. Clearly, diagnosing the error at the implementation level is much more expensive than detecting and fixing the error in the design.

There are many contributions in the literature addressing the verification of UML/OCL diagrams. Nevertheless, the expressivity of UML/OCL is a challenge as it makes consistency checking a computationally complex problem: checking UML class diagrams for consistency is EXPTIME-complete assuming that there are no OCL constraints [BCG05], and it becomes undecidable when OCL invariants are considered (as they are as expressive as full first order formulas) [BCG05]. For this reason, some methods focus on the verification of UML class diagrams without OCL constraints [BCG05, SMSJ03, BM08, CCGM04, MM06, MB07]. Other approaches support OCL but are based on interactive theorem proving and therefore require user assistance to complete the proofs [BW09, KFdB$^+$05]. Finally other tools avoid undecidability by using semi-decidable methods or limit the UML/OCL constructs that may appear in the diagrams [QRT$^+$10, QT12, Jac06, SWK$^+$10, WBBK10, GBR05, AIAB11]. In particular, arithmetic expression support in OCL constraints is a feature which is missing or has shortcomings in most proposals. As a whole, we believe that these limitations impair a wide adoption of formal methods within the MDD community.

1.2. Motivating example

Consistency errors can be inadvertently introduced even in very small UML/OCL models. As an example, consider the simple class diagram of Fig. 1 that will be used as a running example throughout the paper. The diagram models the relationship between researchers and the papers they write (association *Writes*) or review (association *Reviews*).

This class diagram is complemented with a set of OCL expressions that specify additional constraints for the model. For instance, *NoStudentReviewers* states that the referees of a paper cannot be students. In this constraint, the *self* variable represents an arbitrary instance of the context type chosen to define the

constraint, in this case *Paper*. Being an invariant means that the constraint must be true for all possible values of *self*. The expression *self.referee* retrieves the Set of all the researcher objects linked to the paper *self* through the association *Reviews*. Finally, the *forAll* quantification evaluates the *not isStudent* condition on the collection of researchers retrieved by that expression and returns true if all of them satisfy it.

Notice that several expressions in these OCL invariants, such as *self.referee*, are computing collections of objects and operating with them. Some examples of these operations are checking if an object is not included in a Set (operation *excludes* in invariant *NoSelfReviews*), computing the number of elements in the Set (operation *size* in invariant *LimitsOnStudentPapers*), computing the subset of elements satisfying a property (*select* in invariant *LimitsOnStudentPapers*) or checking existential or universal properties on elements of the Set (*forAll* and *exists* quantifiers). As it is shown in this example, OCL collections allow the concise definition of complex properties and they are an important notion in the OCL notation.

Even if perhaps it is not easy to see at first sight, this model is wrong because it does not satisfy a basic correctness property: *strong satisfiability*. A model is strongly satisfiable if it is possible to create at least a valid 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 constraint is violated. Therefore, non-satisfiable models are completely useless since users will never be able to populate them at run-time in a way that all constraints evaluate to true. In particular, this example is unsatisfiable due to two different reasons:

1. The multiplicities of association *Reviews* require exactly three distinct researchers per paper acting as referees, as indicated by the number three next to the *referee* role that the *Researcher* class plays in the association *Reviews*. If we denote by $|X|$ the number of objects of a given class X, this multiplicity means that $|Researcher| = 3 \cdot |Paper|$. Meanwhile, the multiplicities of *Writes* requires one or two researchers per paper acting as authors (multiplicity "1..2" next to the *author* role), and therefore $|Paper| \leq |Researcher| \leq 2 \cdot |Paper|$. 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 multiplicity 1 next to the *manuscript* role in *Writes*), all authors must review papers (multiplicity 1 next to the *submission* role in *Reviews*) and there must be at least one student paper (constraint *LimitsOnStudentPapers*) with a student author (constraint *AuthorsOfStudentPaper*).

We have shown that this simple example does not satisfy a property which is assumed by default by any model designer: that it is possible to build a finite and non-empty instantiation of the model without violating the visual and textual constraints it contains. In addition to this notion of strong satisfiability, another reasonable assumption is that the model does not contain *subsumed*

constraints, i.e. a constraint which can be removed without changing the set of legal instances. A subsumed constraint may be a symptom of an unexpected interaction among constraints or the incorrect definition of some constraints. A more extensive catalog of quality criteria is provided in Section 5. If we do not fix this type of errors in the modeling phase at design-time, developers will waste their time implementing this model in the final technology platform before realizing, when testing the system at run-time, that it contains fundamental errors.

1.3. Contributions of this paper

The main goal of this paper is to present a 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 an *exhaustive search* within the finite solution space. As a drawback, this approach is *incomplete* as potential instances outside of the bounded scope are not considered during the analysis. That means that if no instance is found, the tool will inform the user of the lack of solutions, explaining that in this case the answer is *inconclusive*. We will argue that considering a finite solution space is a reasonable trade-off regarding the features offered by other existing verification methods.

Our method uses the Constraint Programming paradigm [MS98] as an underlying formalism. We have developed 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, such as satisfiability of the model, liveliness of a class, redundancy of a constraint and so forth, can then be checked on the resulting CSP. Our choice of using UML/OCL models as input is based on the wide adoption of the UML within the software community and its high-level modeling constructs, not tied to any particular implementation technology. However, we believe that many of the concepts introduced in the paper can be useful for verifying correctness of models specified with other modeling languages as well, such as Domain-Specific Modeling Languages.

Moreover, in order to improve the usability of our verification method, we have developed a graphical front-end tool called UMLtoCSP [U2C] which hides the underlying analysis process. The input of the tool is a UML class diagram encoded in an XMI or Ecore file format plus (optionally) a text file with the OCL constraints. Meanwhile, the output of the tool is a UML object diagram that proves the property (if it holds). Users of the tool do not need to be familiar with Prolog or CSPs to use the tool: the input and output notations are amenable to UML designers and the entire verification process is completely automated and hidden from the user. In this sense, we follow the paradigm of *hidden formal methods* [Ber02] to improve the usability of the tool and its results.

Preliminary results on this approach were presented in the Model Driven Verification and Validation Workshop [CCR08]. In this paper, we considerably extend those preliminary results with a more comprehensive description of the method and the underlying tool, an improved UML/OCL to CSP mapping strategy, an evaluation of the problem complexity and efficiency results and a more detailed comparison with related approaches.

1.4. Applicability of this approach

The approach presented in this paper has been applied successfully to two different research problems in a related field, model transformation.

[CCGdL10a] considers the analysis of in-place model transformations described using graph transformation rules. Through this analysis, it is possible to validate several properties of the transformation, e.g. identifying rules which are not applicable or conflicts among pairs of rules.

[CCGdL10b] considers the verification and validation of declarative model-to-model transformations, i.e. transformations from one metamodel to another that are described in a non-operational way. Two types of declarative M2M transformation notations are considered: Triple Graph Grammars (TGGs) and QVT-Relations. This process translates the rules of a M2M transformation into a set of OCL invariants that encode the behavior of the transformation. These invariants can be analysed to execute the transformation in both directions (i.e. from source to target or from target to source) or to check properties of the transformation, e.g. "is there a source model with more than one legal corresponding target?".

In both problems, the tool UMLtoCSP is used as the underlying solver to perform verification and validation. The supported UML/OCL subset is sufficient to formalize the problem (encoding transformation rules described in three different notations) and the properties of interest for the model transformation. These contributions provide evidence that this approach can be used to describe interesting models and check relevant properties.

1.5. Paper organization

The rest of the paper is structured as follows. The next section introduces the main steps of our method. Section 3 introduces Constraint Programming concepts and notation. Later, section 4 describes how to transform a UML/OCL model into a CSP. Section 5 presents some correctness properties for verification and validation and their representation as additional constraints in the CSP. The resolution of the generated CSP is shown in section 6. The verification tool that implements our approach is introduced in section 7. Section 8 discusses some efficiency aspects. Previous work and theoretical aspects are analysed in section 9. Finally, section 10 draws some conclusions and highlights future work.

2. Overview of the approach

To determine the correctness of a model, our method follows the procedure depicted in Figure 2. First, the designer provides an input UML/OCL

model, created using an existing UML CASE tool. Then the designer selects the correctness property to evaluate on the model. These correctness properties express the feasibility of creating legal instances of classes and associations in the model (satisfiability properties) and the interactions among different integrity constraints (subsumption and redundancy properties). Next, the model plus the correctness property is translated into a CSP such that the CSP has a solution if and only if the model satisfies the property[2]. The translation process uses our own specialized CSP library encoding the semantics of the UML and OCL constructs in order to simplify the transformation.

The actual evaluation of the CSP is made with a state-of-the-art constraint solver. The results reported by the solver are interpreted and passed back to the user as an object diagram that proves the property (if there is a solution to the CSP) or as a text message informing that the property is not satisfied.

Intuitively, the generated CSP describes the possible set of valid instantiations of the model by using (list) variables that encode the objects and links in the instantiation, the values of the attributes of those objects, etc. The domain of the variables maps the structure and types of the elements in the model. Integrity constraints in the model such as multiplicity constraints or OCL invariants are translated into constraints in the CSP that restrict the legal values for these variables. The correctness properties are represented as additional constraints in the CSP. For instance, satisfiability (non-emptiness of the instantiation) can be imposed as a new constraint: a lower bound on the number of objects and links, i.e. a constraint on the minimum size of the corresponding lists. 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 fails to satisfy the property. The next section provides more information about the search in a CSP.

As an example, the CSP for the running example about Papers and Researchers would roughly consist of four list variables that represent the population of the *Paper* and *Researcher* classes and of the *Writes* and *Reviews* associations. The structure of the elements of each list mirrors the structure of the corresponding model elements. Several constraints restrict the possible number and values of elements in the lists. For instance, constraints will ensure that each paper in the *Papers* list appears at least once in the *Writes* list (all papers must be written by at least a *Researcher* according to the constraints in the model) and that its *wordCount* value is lower than 10000 (as forced by the *PaperLength* constraint). On top of this initial CSP, we need to add the constraints to ensure that the model satisfies the correctness property we are interested in. As an example, when checking for strong satisfiability our method would add a new constraint into the CSP stating that none of the four list variables can be empty.

The analysis of this CSP by the solver would conclude that it is not possible to find a solution since the solver will be unable to create and assign elements

[2]A formal proof of the soundness of this translation is out of the scope of this paper.

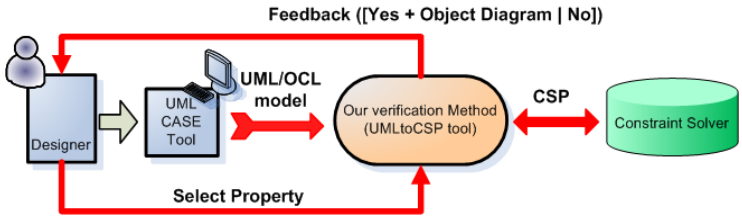


Figure 2: Schema of our method.

to the lists in such a way that (1) all previous constraints are fulfilled and (2) at the same time, the lists are not left empty (forbidden by the constraint imposed by the satisfiability property we are trying to determine). Therefore, we may conclude that this model is not strongly satisfiable.

3. Basic concepts of Constraint Programming

Constraint Programming [MS98, AW07] is a 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.

CP can be used to solve different kinds of problems. In this work it will be used to solve those 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. Typically, most constraints can be defined as equalities ($=$), disequalities ($\neq$) or inequalities ($<, >, \leq, \geq$) of arithmetic expressions over variables, or a boolean combination of such constraints, e.g. $(x = y) \vee (2x^2 \geq 0)$. 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 *unsatisfiable*.

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 available. 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 and avoided, pruning the search tree in this way. These techniques are an effective mechanism to reduce the search space and are implemented by default in most constraint solvers.

$$
\begin{array}{rcl}
V & = & \{\ X,\ Y\ \} \\
D & = & \{\ domain(X) = [-100, +100], \\
 & & \ \ domain(Y) = [-100, +100]\ \} \\
C & = & \{\ X > 20,\ Y \leq 15,\ X = Y\ \} \\
 & & \ \ \downarrow\ \textbf{Propagating constraint}\ (\mathbf{X > 20}) \\
D' & = & \{\ domain(X) = [21, +100], \\
 & & \ \ domain(Y) = [-100, +100]\ \} \\
 & & \ \ \downarrow\ \textbf{Propagating constraint}\ (\mathbf{Y \leq 15}) \\
D'' & = & \{\ domain(X) = [21, +100], \\
 & & \ \ domain(Y) = [-100, 15]\ \} \\
 & & \ \ \downarrow\ \textbf{Propagating constraint}\ (\mathbf{X = Y}) \\
D''' & = & \{\ domain(X) = \emptyset, \\
 & & \ \ domain(Y) = \emptyset\ \}
\end{array}
$$

Figure 3: Constraint propagation example

As an example, consider the simple CSP of Fig. 3. The CSP consists of two variables X and Y whose domain ranges from -100 to $+100$. There are three constraints: $X > 20$, $Y \leq 15$ and $X = Y$. For this example, constraint propagation techniques suffice to directly prove the unfeasibility of the CSP with neither instantiations nor backtracks required. First, the lower and upper bounds for the domains can be tightened by leaving only feasible values inside the domain. Furthermore, in the last step, the fact that the domains for X and Y are disjoint can be used to deduce that $(X = Y)$ is impossible: the set of feasible values in the domain becomes empty ($\emptyset$), so we conclude that the CSP is unfeasible. Typically, constraint propagation is not that successful, but it is an effective mechanism to reduce the search space. Some types of arithmetic constraints, e.g. linear inequalities, have specialised numerical solvers to perform complex propagations among variables.

Without loss of generality, in this paper we will describe CSPs using the syntax provided by the ECL^{i}PS^e Constraint Programming System [The07, AW07]. 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.

The proposed approach can easily be codified in any other Constraint Programming language, as all of them provide support for suspensions, lists and finite domain solvers (see section 4.3 for more information on these topics). Some examples of alternative solvers would be GNU-Prolog [GNU], Oz[Oz], SICStus Prolog[SIC], CHIP V5[CHI], ILOG CP[ILO], JaCoP[JaC], Comet[Com]

or Cream[Cre]. The translation of UML/OCL diagrams into other families of restricted constraint problems, such as SAT or SAT Modulo Theories (SMT), requires completely different encoding strategies which are out of the scope of this paper. These strategies depend on the specific formalism being used, e.g. see [SWK+10] for a translation of UML/OCL into SAT. Every approach has its proper advantages and drawbacks. In our case, we have chosen CP mainly because it is highly expressive and a natural way to code constraints and arithmetic operations.

4. Translation of UML/OCL Class Diagrams

This section describes the transformation of a UML/OCL 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. However, the automatic computation of these (smallest) domains from the constraints in IC is undecidable, even though it can be approached from the static analysis of OCL invariants [YBP07]. This problem will not be addressed in this paper, assuming instead that domains 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. Note that some of the constraints generated by our method in the CSP are implicit in the semantics of UML but must be made explicit in the CSP. For example, we need to state explicitly that all instances of a class are also instances of its superclasses. In [GR02] a translation of all these graphical constraints into an OCL representation is proposed.

Section 4.1 describes the translation of the UML elements in the model while section 4.2 focuses on the translation of the OCL integrity constraints. Both parts rely on our UML/OCL CSP library, introduced in section 4.3. The library extends predicates available in the Prolog dialect used by $\mathsf{ECL^iPS^e}$ with a new set of predicates that map the semantics of the predefined OCL operations. Rather than hard-coding these OCL operations in the translation procedure, we have decided to group them in a separate library to simplify the translation.

4.1. Transformation of UML constructs

To illustrate the translation of UML constructs, we will refer to the example in Figure 4 throughout this section. Figure 4(a) shows a class diagram and

Figures 4(b) and (c) show the translation of this diagram into variables and constraints of a CSP, plus a potentially legal instantiation.

4.1.1. 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[3]. 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 $at \in c.ownedAttribute$[4].

  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 require at least a lower and upper bound for the attribute. 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*: $\forall x, y \in Instances_c : x \neq y \rightarrow x.oid \neq y.oid$

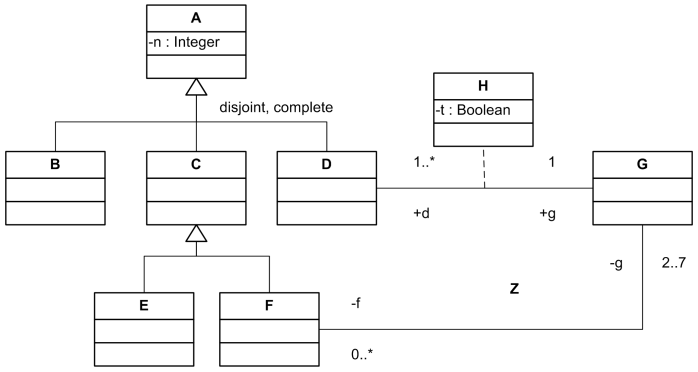
For example, figure 4(b) illustrates the structure of the $Instances$ variable for classes A to G, where the constraints on the number of instances and the uniqueness of oids within a class are omitted for brevity. Notice how even in classes without attributes, it is still necessary to keep track of the *oid*, for example, to keep track of inheritance relations or the participation of each object in associations.

4.1.2. 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:

[3] Sets of instances are defined as Prolog lists with additional constraints to avoid duplicates.

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



(a)

A (oid_A, n)	**B** (oid_B)	**C** (oid_C)	**D** (oid_D)	**E** (oid_E)	**F** (oid_F)	**G** (oid_G)	**H** (oid_H, t, d, g)	**Z** (f, g)
$(1,10),(2,14),$ $(3,14),(4,20),$ $(5,90),(6,20),$ $(7,10)$	(1)	(2), (3)	(4), (5)	(2)	(2)	(1), (2)	$(1, true, 4, 1),$ $(2, false, 5, 2)$	$(2,1),$ $(2,2)$

(b)

Constraints on association Z

Existence of referenced objects	$\{Z.f\} \subseteq \{oid_F\}, \{Z.g\} \subseteq \{oid_G\}$
Bounds on cardinalities	$(Size_Z \leq Size_F \cdot Size_G),$ $(2 \cdot Size_F \leq Size_Z \leq 7 \cdot Size_F)$
Multiplicities of role "g"	$\forall x \in \{oid_F\} : 2 \leq \#\{l \in Z : l.f = x\} \leq 7$

Constraints on association class H

Existence of referenced objects	$\{H.d\} \subseteq \{oid_D\}, \{H.g\} \subseteq \{oid_G\}$
Bounds on cardinalities	$(Size_H \leq Size_D \cdot Size_G), (Size_G \leq Size_H),$ $(Size_D \leq Size_H \leq Size_D)$
Multiplicities of role "g"	$\forall x \in \{oid_D\} : 1 \leq \#\{l \in H : l.d = x\} \leq 1$
Multiplicities of role "d"	$\forall y \in \{oid_G\} : 1 \leq \#\{l \in H : l.g = x\}$

Constraints on generalisation set A-B-C-D

Number of instances	$(Size_A \geq Size_B), (Size_A \geq Size_C), (Size_A \geq Size_D)$
Existence of oids in supertype	$\{oid_B\} \subseteq \{oid_A\}, \{oid_C\} \subseteq \{oid_A\}, \{oid_D\} \subseteq \{oid_A\}$
Disjointness (cardinalities)	$Size_A \geq Size_B + Size_C + Size_D$
Disjointness (oids)	$\{oid_B\} \cap \{oid_C\} = \{oid_B\} \cap \{oid_D\} =$ $= \{oid_C\} \cap \{oid_D\} = \emptyset$
Completeness (cardinalities)	$Size_A \leq Size_B + Size_C + Size_D$
Completeness (oids)	$\{oid_A\} = \{oid_B\} \cup \{oid_C\} \cup \{oid_D\}$

Constraints on generalisation set C-E-F

Number of instances	$(Size_C \geq Size_E), (Size_C \geq Size_F),$
Existence of oids in supertype	$\{oid_E\} \subseteq \{oid_C\}, \{oid_F\} \subseteq \{oid_C\}$

(c)

Figure 4: Example of the translation of UML class diagram constructs: (a) class diagram, (b) corresponding variables in the CSP with a possible legal instantiation and (c) a selection of corresponding constraints 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 collection 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* [Obj11] of the association is set to false.

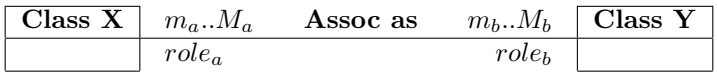
- *Bounds on cardinalities*: The multiplicities of an association impose constraints on the number of instances of the participant classes and the association. The explicit representation of these constraints in the CSP is presented in Fig. 5 for binary associations.

- *Multiplicities of the association*: Multiplicity constraints must also be satisfied by each individual object of the participant classes. For n-ary associations, several multiplicity constraints among participant objects may be defined [Obj11]. In particular, for binary constraints the following constraint must hold:

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

Figure 4(b) illustrates the *Instances* variables for a binary association Z: it keeps track of the oids of the participating objects from classes F and G. Then, Figure 4(c) shows relevant constraints added to the CSP for this association, assuming that *isUnique* is set to true. The notation $\{Z.f\} \subseteq \{oid_F\}$ is a shorthand denoting that all oids referenced from role f in $Instances_Z$ should correspond to one of the oids in $Instances_F$. Regarding the rest of constraints, it should be noted that there is a role with multiplicity "0..*", i.e. any number of participating objects. This multiplicity does not generate any constraint for this role in the CSP.

Associations that are not referenced (i.e. navigated) in any constraint and that do not state any multiplicity constraint (all participants have a "0..*"



$$Size_{as} \leq Size_X \cdot Size_Y$$
$$m_a \cdot Size_Y \leq Size_{as} \leq M_a \cdot Size_Y$$
$$m_b \cdot Size_X \leq Size_{as} \leq M_b \cdot Size_X$$

Figure 5: Implicit cardinality constraints due to the association multiplicities [CCGM04]

multiplicity) can be discarded during the translation process. The population of those associations does not affect the existence of solutions to the CSP.

Compositions and aggregations are just two special kinds of associations and thus are translated following the procedure explained in this section complemented with the translation of the OCL constraints needed to enforce their specific *containment* or *whole-part* semantics. These constraints are taken from [GR02] and added to the pool of OCL constraints of the model translated as explained in the next section.

4.1.3. 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. More specifically, variables for association classes are $Instances_{ac}$ and $Size_{ac}$ where the structure of elements in $Instances_{ac}$ includes all fields corresponding to the transformation of the class facet of ac plus the fields corresponding to the transformation of the association facet of ac. Likewise, constraints for ac are the combination of constraints for classes and for associations. Therefore, this transformation considers the special semantics of association classes [Obj11, GR02] stating that each instance of the association class should correspond to a link in the underlying association.

For example, Figures 4(b) and (c) illustrate the variables and constraints for the associative class H from our example. Notice that the structure of the $Instances$ variable includes oids and attributes like objects, and also role names like associations.

4.1.4. 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.

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*: $Size_{sub} \leq Size_{sup}$

```
% Unconstrained attributes are not in the CSP
:-local struct researcher(oid,isStudent).
:-local struct paper(oid,wordCount,studentPaper).
:-local struct reviews(submission,referee).
:-local struct writes(manuscript,author).
```

Figure 6: Structs for the translation of classes and associations of the running example.

- *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 $S_1..S_n$:
  - $Size_S \leq \sum_i Size_{S_i}$
  - $\forall o_1 \in Instances_S : \exists i \in [1, n] : \exists o_2 \in Instances_{Si} : o_1.oid = o_2.oid$

For example, Figure 4(c) describes the set of constraints involved in two different generalization sets: the subclasses of A and the subclasses of C. Also, Figure 4(b) shows an instantiation of these classes which helps to illustrate the role of oids through inheritance: an object preserves the same oid in the subclass and the superclass. Even though this approach does not support multiple inheritance, it is able to describe complex inheritance scenarios. For example, it supports *overlapping* inheritance, i.e. the same oid is used in two or more subclasses of the same superclass, and also *complete inheritance*, i.e. all oids from the superclass must be used in at least one of the subclasses.

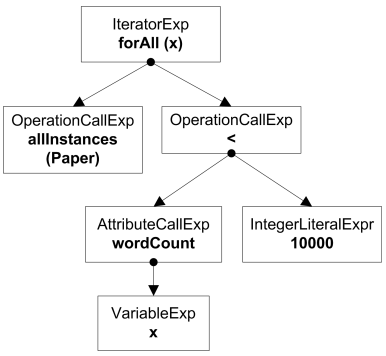
4.1.5. Transformation of the running example

Before describing the translation of OCL constraints, we retake our running example from Figure 1 and illustrate the translation process described so far. This time we introduce the syntax of the $\mathsf{ECL^iPS^e}$ code for the CSP, which will be used extensively in the following section.

Figure 6 illustrates the *Instances* variables for the classes and associations in the running example. Notice that some attributes such as *name* from class *Researcher* or *title* from class *Paper* do not appear in the *Instances* variables to improve the efficiency of the solver. Regarding the constraints for the UML constructs in this diagram, they primarily involve the multiplicities of associations *Writes* and *Reviews*. Some example predicates would be `differentOids`, which checks that all oids of a single class are different, or `linksConstraintMultiplicities`, which ensures that the oids of participants in a binary association preserve the multiplicities of each role.

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

4.2. Translation of OCL constraints

Integrity constraints in OCL [Obj10] 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. 7 shows an example of the translation process presented in this section. This example has been simplified to improve the legibility and understandability of the constraints.

An OCL constraint can be viewed as an instance of the OCL metamodel with a tree shape, with a close resemblance to the abstract syntax tree of the textual constraint that would be constructed by an OCL parser. For instance, the simplified tree representation for *PaperLength* constraint is illustrated in Fig. 7. Leaf 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 outermost operation of the constraint. Packages like the Dresden OCL toolkit [Dem04] can parse textual OCL constraints and build the corresponding trees.

As a preliminary step and to homogenize the translation procedure, we express all constraints in terms of the *allInstances*[5] operation using the following expansion rule:

context T **inv**: B $\quad\Rightarrow\quad$ **context** T **inv**: T::allInstances()−>forAll(v|B')

where B' is obtained by replacing all occurrences of *self* in B with v. This step serves two purposes. First, it encodes the semantics of an OCL invariant, a property that should evaluate to true for all objects of a context type, using the constraint language of the CSP. And second, it eliminates the variable *self* as a special case to be treated separately during the translation, as it becomes a quantifier variable like any other.

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 $\mathsf{ECL^iPS^e}$ Prolog compound term with a *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 all *instances* for each class and association. The i-th position of this list holds all the instances of class/association i, i.e. it holds the contents of the corresponding $Instances_i$ variable defined in section 4.1. 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.

[5] allInstances is a predefined OCL operation that returns the set of instances of the type.

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 kind of operation applied in the node.

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 (according to the semantics of the OCL operation represented by the node) in terms of the results of its subexpressions. Basic types (e.g. boolean or integer) and basic OCL operations (e.g. logical and arithmetic) have a direct counterpart implementation in the ECL^{i}PS^e constraint libraries. For more complex operations, such as iterators or operations on Collections, we have developed a new ECL^{i}PS^e library (see the next section) that implements the operations defined in the OCL Standard Library [Obj10]. Nevertheless, for the sake of simplicity, in Fig. 7 we have directly added to each node the required computation without relying on our external library.

As a final step, once the translation for the body expression has been completed, we add to the CSP a new constraint representing the original OCL invariant. This constraint is defined as:

```
nameConstraint(Instances):- rootNode(Instances,[],Result), Result#=1.
```

In other words, the constraint is true when the `rootNode` evaluates to true. For example, see the *paperLength* constraint in Fig. 7(c).

4.3. UML/OCL Prolog Library

OCL provides a rich set of predefined types (e.g. collections: sequences, sets, bags,...), operations (arithmetic, logic, set,...) and iterator expressions (for all, exists, iterate,...) for the definition of complex constraints as part of the OCL Standard Library. In order to analyze OCL constraints, it is necessary to provide a translation for those operations in terms of the Prolog-based language used by the ECL^{i}PS^e solver. Rather than hard-coding these operations in the tool translation procedure, we have decided to group them in a separate library. When translating the OCL constraints, we will call the appropriate library predicate to implement the specific semantics of each constraint node. For example, the library predicate `ocl_int_equals` will be reused every time a constraint includes an integer equality comparison between two subexpressions. This separation provides additional flexibility as changes in the implementation of the OCL operators can be performed without modifying the translation method and thus, without changing the source code of UMLtoCSP[6].

[6]In fact, it is not even necessary to recompile the tool, as Prolog is an interpreted language.

The Prolog dialect used by ECL^{i}PS^e is an extension of pure Prolog. A full description of this notation is available in [AW07]. We will simply mention those extensions required for the comprehension of the implementation of our UML/OCL library:

- Support for higher-order predicates, i.e. the ability to pass predicate names as arguments.
- Syntactic flavor to facilitate the definition of constraints, e.g. iterator constructs like `for` or `foreach`. These constructs are specially useful to operate with Prolog lists, so they are used in collection operations, OCL iterators and navigations.
- Support for the definition of *suspended* constraints, i.e. delaying the execution of a predicate whose arguments have not been assigned yet.

The first extension, higher-order predicates, is provided by a library called `apply`. With this library, it is possible to pass predicate names as arguments of other predicates, and invoke those predicates later with a list of arguments that is constructed at run-time. Intuitively, it is possible to build generic predicates like "evaluate this predicate in each element of a collection", i.e. the "collect" function of collections. This function is computed by a predicate with the following signature: `ocl_set_collect(Instances, Vars, Set, Predicate, Result)`, where `Set` is the collection where the operation must be applied, `Predicate` is the name of the predicate to be applied and `Result` is the result of the operation. It is possible to implement iterators like "exists" or "forAll" in a similar way.

In the following sections, we will describe some design decision and implementation details of the UML/OCL Prolog library (available at [U2C] as a part of the download of the tool UMLtoCSP). We will discuss separately the implementation of basic types (subsection 4.3.1), issues with suspensions (subsection 4.3.2), the implementation of OCL collections (subsection 4.3.3) and the OCL iterator expressions (subsection 4.3.4).

4.3.1. OCL Basic Types

ECL^{i}PS^e provides several solver libraries, each one targeting a different type of constraints, e.g. linear constraints, graph constraints, ... In UMLtoCSP, we use the *finite domain interval constraint solver* library (`ic`), which analyses constraints by considering the domain of each variable as one or more finite intervals of values. This library provides support for integers and boolean[7] variables, and defines all the usual arithmetic ($+, *, /, \%$), relational ($=, \neq, <, >, \leq, \geq, \min, \max$) and boolean ($\wedge, \vee, \rightarrow$) operators. Furthermore, those operators which are not provided by the library itself, e.g. xor, can be simply defined using the Prolog language. Thanks to these predefined operators we can easily implement the support for OCL basic types in our library.

[7] Boolean values are represented as integers whose value is either zero (false) or one (true).

For example, let us consider the implementation of the relational operator "greater-than" for comparing two integers. This operator takes as parameters two predicate names and stores the result of the comparison in a variable called Result. The ECL^{i}PS^e code for this operation is the following:

```
ocl_int_greater_than(Instances, Vars, Pred1, Pred2, Result) :-
   apply(Pred1, [Instances, Vars, X]),  % X is the result of evaluating Pred1
   apply(Pred2, [Instances, Vars, Y]),  % Y is the result of evaluating Pred2
   Result::0..1,                        % Result is a boolean value
   #>(X, Y, Result).                    % Result is true iff X > Y
```

This predicate uses the predefined operator `#>` which is provided by the predefined `ic` library to define the integral "greater-than" constraint. All integer operators in the library have the "`#`" prefix. Whenever there is a potential ambiguity, e.g. between the default Prolog language operator and the operator redefined by the library, the name of the library is prepended to the operator. For example, the logical "and" operator from the `ic` library is invoked as `ic:and` to distinguish it from the default Prolog "and".

Note that in this predicate we are not forcing X to be greater than Y, we are just evaluating whether this is true and storing the result in the *Result* variable. This way of using boolean constraints is known as *reified constraints*.

A disadvantage of using the `ic` library is the lack of support for constraints on strings. In fact, there is no ECL^{i}PS^e library which provides efficient support for the type of string constraints that can be written in OCL, e.g. substrings or concatenations. In models with attributes of type string that are only compared among them with no substrings and concatenations, strings can be encoded as an integer or enumerated attribute. However, our current implementation does not provide support for other string operations in OCL constraints.

4.3.2. Suspensions

The result of an operation cannot be evaluated until the values of its arguments are known. In the constraint programming paradigm, the constraints of the CSP are defined in the beginning, and then we try to assign values to variables in such a way that all constraints are satisfied. It should be noted that constraints are defined before the values of variables are available, so we need the concept of *suspended* constraint, a constraint which has been defined (e.g. $a = b$) but cannot be evaluated until the values of variables are available (e.g. until we have tried to assign a or b).

A possible solution is delaying the evaluation of a constraint until the complete assignment is available. However, this conservative approach is very inefficient. Instead, it might be possible to detect that an assignment violates a constraint without assigning values to all variables. For example, if we have variables a to z and a constraint $a = b$, the constraint can be evaluated as soon as a and b are assigned. Waiting for the other variables to become assigned means evaluating the constraint for all possible combinations of variables c to z, a number which grows exponentially with the number of variables involved.

To avoid inefficiencies due to the late evaluation of constraints, ECL^{i}PS^e provides a mechanism to *suspend* constraints when they are defined and *wake* them whenever there is a possibility to propagate information between variables. For instance, constraints are awakened when *one* argument is assigned (and not necessarily *all*). The constraint propagation built in ECL^{i}PS^e can sometimes infer information about the result without knowing all the arguments of an operation. For example, if one argument of a product is 0, the result is automatically zero regardless of the other argument. The same type of early evaluation can be applied to boolean operations. Sometimes it is also possible to use the domain of an argument to infer information about the domain of other arguments. For example, in $a = b$ if we assign a the value 7, two situations can happen: if the value 7 is within the domain of b, then we know that the equality holds and $b = 7$; otherwise, we know that the equality does not hold.

In our library, we use two types of suspensions, that of the predefined predicates and that of the new rules that we define:

- In the predefined constraints of ECL^{i}PS^e libraries, like `#>`, or `ic:and`, suspension is already built in. This means that we can write `Value1 #> Value2` and ECL^{i}PS^e handles the suspension transparently.

- In the new rules that we define, it is necessary to specify conditions that restrict when it cannot be evaluated and must become suspended. This is achieved using the declarative suspension clause `delay-if` before the definition of a rule. For example, the clause `delay ocl_col_size(X) if var(X)` delays the execution of the rule `ocl_col_size` until its argument ceases being an unassigned variable. The constraint will be automatically woken and evaluated by the ECL^{i}PS^e solver when variable `X` is given a value.

In this second group, it is very important to wake constraints as soon as possible: if the current solution is unfeasible, discovering it early will avoid unnecessary backtracking and greatly reduce the execution time. However, it is possible to define suspensions which are too conservative, for example, in collection operators. An operation like nonEmpty() can be evaluated as soon as we know that the list has no elements or more than one, it is not necessary to wait until the specific value of the elements of the collection is available. Selecting the right degree of suspension for each operation has been an important task in the design of the library.

4.3.3. OCL Collection Types

Lists are a key concept of the ECL^{i}PS^e notation. These are represented as a sequence of elements separated by commas and enclosed between brackets, e.g. `[]` or `[2, 7, 25]`. They admit the repetition of elements and the order among elements matters. Formally, a list is defined recursively using the empty list (`[]`) and the constructor `|` which appends one element (*head*) to the beginning of an existing list (*tail*), e.g. `[head | tail]`. For example, the previous notation for lists is a shorthand of:

[2 | [7 | [25 | []]]]

Lists will be used as the backbone for the representation of OCL collections: all the elements of an OCL collection will be stored in a Prolog list. The Prolog semantics of lists matches that of OCL sequences, so the implementation of sequence operations will be straightforward. For example, the implementation of operation "size()" of collections, which returns the number of elements in a collection, relies on the "length" operation in Prolog lists:

```
ocl_col_size(Col, Size) :- length(Col, Size)
```

Similarly, other operations rely on Prolog operations on lists.

However, we needed to code additional predicates/constraints to manage other OCL collection types, i.e. to correctly represent their semantics when stored as Prolog lists, e.g. to enforce the uniqueness of elements in a set collection type. These additional predicates/constraints are the following:

Sets: Additional checks are required to ensure that the set contains no duplicates after the insertion of new elements (OCL operation "including") and the union of two sets or a set and a bag (union). To improve the efficiency of this representation, the elements in the Prolog list are kept ordered at all times.

Bags: As lists allow duplicate elements, bags can be represented directly as lists much like sequences. However, the performance of several operations like intersection or the equality check, can be improved if the elements of the bags are ordered. Rather than keeping the elements ordered at all times, the elements are ordered on demand when the equality check or intersection operations are invoked.

Ordered sets: The operations on ordered sets are equivalent to those of a sequence, except for the check to avoid duplicate elements. In this collection, elements cannot be stored in an ordered list as the insertion order must be preserved.

Operations on collections can be classified into two categories according to the degree of suspension that they require. In the first category, there are operations on collections which can be evaluated when the number of elements of the collections is known, even if the specific values of the elements in the collection is unknown, e.g. size(), isEmpty() and most operations on empty collections. Such operations must be delayed until the size of the collection is known. In a second category, other operations need that the values of the elements in the collection are known a priori, e.g. includes() in a non-empty list. In this case, it is not possible to evaluate the operation until all the elements of the collection have been given a value.

4.3.4. Iterators

OCL provides a set of iterator expressions over collections, e.g. existential (exists) and universal (forAll) quantification. As OCL collections are encoded in Prolog lists, the iterators must be translated in terms of lists.

The core operation of iterators is the ability to evaluate an OCL expression in each element of a collection. Talking in Prolog terms, this translates to the ability to apply a predicate (encoding the OCL expression) to each element of a list (encoding the OCL collection). Using the library `apply` it is possible to obtain a generic implementation of this operation by passing the predicate name to be evaluated as a parameter.

For example, let us consider the following existential quantification:

$$Col\ {->}exists(x|Expr(x))$$

It is possible to evaluate this quantifier in the following way: evaluate $Expr$ in each element x of the collection Col and then count the number of $Expr(x)$ that evaluate to true. The quantifier evaluates to true if and only if that number is greater than zero. This implementation in terms of a CSP requires the following variables and constraints:

- **Variables**:
  - $Result$, a boolean variable which stores the result of the existential quantification.
  - An auxiliary variable N, which is an integer ranging from 0 to the number of elements in the collection.
  - A variable $Expr(x)$ in the CSP for each element x in the collection, i.e. a boolean variable which stores the result of evaluating the expression on x.
- **Constraints**:
  - The necessary constraints to define the value of each $Expr(x)$.
  - A new constraint: $N = \sum_{x \in Col} Expr(x)$.
  - A new constraint: $Result = (N > 0)$.

Figure 8 illustrates the Prolog code required to compute the existential quantification. The auxiliary predicate `property_sat_count` is also used in the implementation of other operators such as *forAll* (universal quantification) or *one* (checking if a property holds for just one element of the collection). This code reuse is shown in Fig. 9. Notice how it is only necessary to change the number of elements which should evaluate to true: in "one" we need only one element, in "forAll" we need all the elements of the collection (we compute the number of elements of the collection using the predicate `ocl_col_size` from the collections).

Using these functions in our translation results in a more compact translation into Prolog. For example, the translation of the node `nodeForAll` in Fig. 7 would become the following:

```
ocl_col_exists(Instances, Vars, Collection, Predicate, Result ) :-
   % N = # of elements where Predicate evaluates to true
   property_sat_count(Instances, Vars, Collection, Predicate, N),
   #>(N, 0, Result).              % Result = (N > 0)

% Count the number of Predicates that evaluate to true in the Collection
property_sat_count(Instances, Vars, Collection, Predicate, Result ) :-
   % Apply Predicate to all elements of Collection
   % Store the results in the list TruthValues
   property_apply(Instances, Vars, Collection, Predicate, TruthValues),
   Result #= sum(TruthValues).      % Result is the sum of all the truth values

% Apply Predicate to each Element of Collection,
% All outputs are stored in the list Result
property_apply(Instances, Vars, Collection, Predicate, Result) :-
   ( foreach(Elem, Collection),   % One Value per Elem in the Collection
     foreach(Value, Result),      % Result is a list of those Values
     param(Predicate, Instances, Vars)
     do
       % Apply Predicate to Elem (Elem is added to the list
       % of visible variables within Predicate)
       apply(Predicate, [Instances, [Elem|Vars], Value]) ).
```

Figure 8: Code for the implementation of the existential quantification (code related to suspensions has been removed for clarity).

```
nodeForAll(Instances, Vars, Result) :-
   nodeAllInstances(Instances, Vars, L),
   ocl_col_forAll(Instances, Vars, L, nodeLessThan, Result).
```

Finally, it should be noted that it is not necessary to manually add optimizations like "the result of an existential quantifier is false when the collection is empty" as they are already considered by the constraint propagation process. If the collection is empty, `property_sat_count` receives an empty list. The operation `sum` directly returns 0 for empty lists. Thus, N will be 0, so $N > 0$ will be false, i.e. the result of the iterator will be false.

4.4. Limitations of the translation from UML/OCL into CSPs

UML and OCL are complex standards which include a large catalog of constructs. As a result, the approach described in this paper does not provide support for all the features described in these standards. The following is a list of unsupported features: multiple inheritance; recursive OCL queries which are infinitely recursive in some model instance and OCL constraints on strings, other than equality tests. In particular, infinite solutions are an inherent limitation of the proposed approach, as it is restricted to bounded verification. With respect to the other constructs, the limitations are more pragmatic: a potential encoding in the CSP has been studied, but as it can reduce the performance of the verification process they have not been included in the approach.

```
ocl_col_one(Instances, Vars, Collection, Predicate, Result ) :-
   property_sat_count(Instances, Vars, Collection, Predicate, N),
   #=(N, 1, Result).            % Result = ( N = 1 )

ocl_col_forAll(Instances, Vars, Collection, Predicate, Result ) :-
   property_sat_count(Instances, Vars, Collection, Predicate, N),
   ocl_col_size(Collection, S),  % S = Number of elements in the collection
   #=(N, S, Result).            % All elements should evaluate to true (N = S)
```

Figure 9: Code for the implementation of other existential quantifications (code related to suspensions has been removed for clarity).

Regarding other constructs that can expressed in different ways in UML/OCL, in some cases our prototype tool has a preferred format that must be respected by designers. For example, our implementation assumes that all attributes in the diagram have a basic type and a multiplicity of 1: otherwise, the designer has to reexpress these attributes of complex types or with other multiplicities by means of an equivalent association (between the class owning the attribute and the type) in the diagram.

Furthermore, our library provides limited support for object-like operations on OCL basic types and collections, as a side-effect of their trivial encoding as Prolog basic atoms and lists. For example, it is not possible to use the operation allInstances() on class Integer or any collection like Set. Also, constraints regarding the type of real, integer, string or collection expressions are not supported, i.e. oclIsTypeOf, oclIsKindOf or oclAsType cannot be applied to these expressions. As a consequence, this affects the implementation of two collection operations which require knowledge about the type of its operands: flatten (replacing all collections within a given collection by the elements they contain) and sum (adding all elements of the collection which are of a numeric type). These two operations are only supported when all elements of the collection are themselves collections (flatten) and when they are all integers (sum), respectively.

Finally, an important difference in the operation of our tool with respect to the OCL standard is the treatment of undefined values. The OCL standard mandates that expressions whose value is unknown must produce an undefined value, e.g. accessing the first element of an empty Sequence. Expressions where any operand is undefined should propagate the undefined value, with the exception of boolean connectives that can short-circuit the evaluation, e.g. $false \wedge undefined = false$. Contrary to the standard, the (pragmatic) behavior taken in this approach is to provide a warning about any operation which would result in an undefined value, stopping the analysis. This comes from the belief that any constraint producing an undefined value deserves further inspection (to make sure that was the intended behaviour by the designer when specifying the constraint) and could (and, possibly, should) be rewritten into an equivalent constraint without undefined subexpressions. Otherwise, undefined values may mask the real result of an OCL expression or they may remain undetected due

to the short-circuit in boolean connectives, leading to unexpected problems in the final software implementation. Besides, there are several inconsistencies in the standard regarding the treatment of undefined and null values which have been pointed out in [BKW10].

5. Quality criteria for UML/OCL class diagrams

In addition to the elements of the model (classes, associations, constraints, ...), it is necessary to encode in the CSP an additional element: the goal of our analysis, i.e. the type of instance of the model that the solver should try to construct. Depending on the selected goal, it is possible to verify or validate different characteristics of our model. In the remainder of this section, we introduce several goals for the analysis of UML/OCL models.

5.1. Correctness properties for verification

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.

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 currently under consideration (and the additional constraint they impose on the CSP) is the following:

Strong satisfiability: The model must have a finite instantiation where the population of all classes and associations is at least one.
Formally: $\forall x \in (Cl \cup As) : Size_x > 0$.

Weak satisfiability: The model must have a finite instantiation where the population of at least one class is at least one.
Formally: $\sum_{x \in Cl} Size_x > 0$

Liveliness of a class c: The model must have a finite instantiation where the population of c is non-empty.
Formally: $Size_c > 0$

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.
Formally: $C_1 \wedge \neg C_2$

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.
Formally: $(C_1 \wedge \neg C_2) \vee (C_2 \wedge \neg C_1)$

Notice that there is a relationship among some of these correctness properties, e.g. strong satisfiability implies weak satisfiability and the lack of constraint subsumption among two constraints implies that none of them is redundant. Checking properties with different degrees of granularity improves the feedback provided to designers. For example, we are able to detect that two constraints are equivalent or that one is stronger than another one: both pieces of information can help designers in the revision of the constraints of the model.

Regarding the satisfiability properties, similar notions have been defined in the literature. The difference among each notion depends on whether (a) it refers to a specific class or all classes in a diagram, (b) it accepts empty instances and (c) it accepts infinite instances. For example, in [ACIG10] weak satisfiability is called *diagram satisfiability*, liveliness of a class is called *satisfiability of a specific class* and strong satisfiability is called *full satisfiability*. Other works such as [MB07] define the notions of *consistency* and *finite satisfiability*. A class is called consistent if and only if it has a legal non-empty instance (possibly infinite), and it is called finitely satisfiable if one of its legal instances is also finite. These properties can be extended to all classes of the diagram simultaneously with *full consistency* (there is an instance where the population of every class is non-empty, but possibly infinite) and *full finite satisfiability* (there is an instance where the population of every class is non-empty and finite). Notice that strong satisfiability as defined in this paper and full finite satisfiability are completely equivalent, as well as liveliness of a class and class finite satisfiability.

From a broader point of view, these correctness notions consider the *intra-model semantic consistency* of UML class diagrams, according to the classification of the surveys [LMT09, MDN09]. This approach is not addressing inter-model consistencies, i.e. the relationship among the different diagrams modeling the same system. We are also not considering other quality notions such as the completeness of the class diagram with respect to the domain or its comprehensibility, among others [MDN09]. Furthermore, this paper does not consider dynamic properties that may appear in the definition of OCL operation pre-conditions and post-conditions, such as the executability of operations or the reachability of a specific system state, e.g. [CCR09, QT09, Jac06].

5.2. Model Validation

Apart from verifying that the model satisfies the previous correctness properties, 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, to validate the behaviour of the model in those situations. The declarative nature of our approach allows the definition of additional constraints that characterise these desired states. For example, in our running example, a designer may want to check whether it is possible to find an instantiation where there is one paper with two student authors. An invariant describing this property is the following:

```
context Paper inv PaperByTwoStudents:
  Paper::allInstances−>exists( p | p.authors−>size() = 2 and
  p.authors −>forAll(r | r.isStudent) )
```

After adding this invariant to the list of integrity constraints of the model, all instances generated by the solver will fulfill this partial specification. Therefore, this method can also be used to perform validation of specific scenarios for the model.

6. Generating the final CSP

The final CSP is obtained as a combination of the translation excerpts generated using the rules of section 4 (for the transformation of the UML/OCL diagram) and section 5 (for the definition of the quality properties to be verified). Remember that this generated CSP has a solution if and only if we can determine that the model satisfies the selected quality properties.

For efficiency reasons (more on this on section 8), the CSP is organized in two subproblems:

1. The *Structural* subproblem. In this first subproblem 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 plus the constraints corresponding to the correctness properties we want to check. In this phase, the goal is to find a legal assignment of values to these $Size_x$ variables. Each legal assignment represents the size dimensions (i.e. number of instances of each class and association) of a possible correct instantiation of the model. This subproblem helps to filter incorrect cardinality assignments that would always result in invalid instantiations, regardless of the actual values given to the association and attribute variables. Therefore, if no assignment is possible at this phase (e.g. due to design errors in the definition of multiplicity constraints), the CSP is directly unfeasible. Instead, if this first subproblem is indeed satisfiable we cannot yet guarantee the correctness of the model, it could happen that we find it impossible to construct a possible legal instantiation of that size due to inconsistencies in the OCL constraints of the model. The definition of this subproblem is similar to the previous work from [CCGM04].
2. The *Global* subproblem. In this 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, i.e. the goal is trying to construct a valid instantiation with exactly $Size_i$ elements for each element i. 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 *Structural* phase, we work on cardinality variables ($Size_x$), while in the *Global* subproblem we are interested in the set of instances ($Instances_x$) of classes and associations.

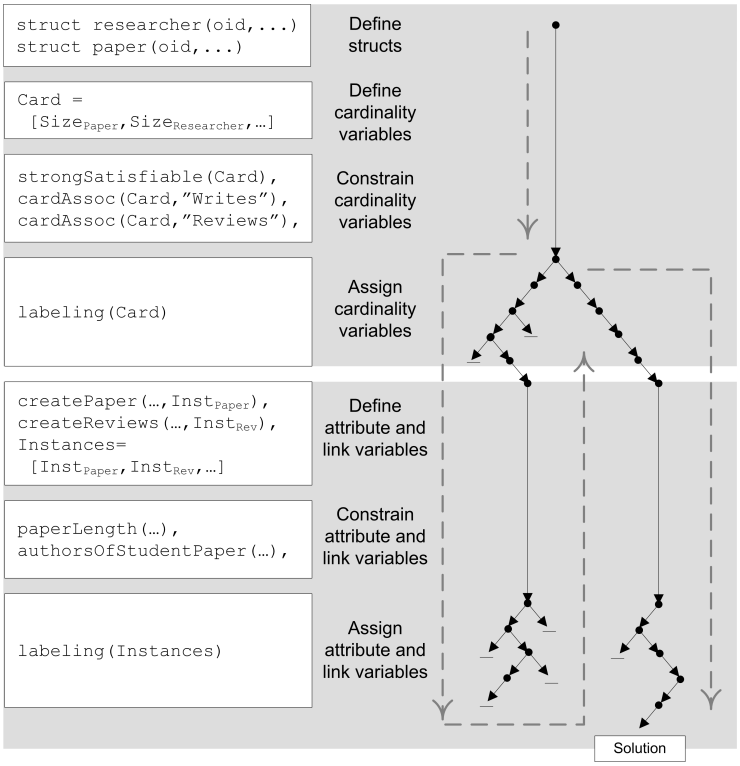


Figure 10: Definition of the CSP for the running example showing the two subproblems, the *Structural* subproblem (upper part) and the *Global* subproblem (bottom part)

As an example, Fig. 10 depicts the CSP corresponding to a satisfiable version of our running example[8]. 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 *Structural* subproblem is found, e.g. one paper, three researchers, one "writes" link and three "reviews" links. However, it is not possible to complete the *Global* 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, e.g. the same solution with two "writes" links instead of one.

Although this two-step search strategy might penalise the overall efficiency of the process in certain cases (see section 8), it has been chosen because of two reasons. First, some CSPs, as the one corresponding to our running example, can be immediately determined as unfeasible when considering the *Structural* subproblem. Second, in the *Global* subproblem we limit the search to cardinality values satisfying the first one, avoiding irrelevant verifications. This means that we can avoid instantiating classes and associations for scenarios in which we know for sure that the cardinality constraints do not hold. However, a disadvantage of this approach is that some constraints in the *Global* subproblem may affect the cardinality of the classes and associations. For example, let us consider an OCL constraint of the form "T::allInstances()−>size() = 7", stating that there are 7 objects of type T. This constraint would appear in the second subproblem, even though it should be constraining the appropriate $Size_T$ variable. Other constraints on the size of classes and associations are not so trivial, like "T::allInstances()−>exists(...)", which requires there must be at least one object of type T. In these scenarios, the search will have unnecessary backtracks, as some solutions provided by the *Structural* phase may not have any feasible solutions in the *Global* phase. A way to avoid these redundant backtracks and therefore improve the search is to reveal these implicit size constraints appearing in OCL expressions using static analysis [YBP07], e.g. $Size_T = 7$ and $Size_T \geq 1$ in the previous examples. These additional size constraints could be added to the first CSP to avoid getting unfeasible solutions caused by the OCL expressions.

7. Tool Implementation

Our prototype tool UMLtoCSP [U2C] is implemented as a set of ECL^{i}PS^e constraint libraries (2000 LoC) and Java classes (11500 LoC) implementing the GUI, the UML/OCL to CSP translation and glue code. The tool also uses several

[8] Fig.1 becomes satisfiable if the multiplicities of *manuscript* and *submission* are changed to 0..1. This version of the model is used to illustrate a successful search. There is more information about this example in section 7.

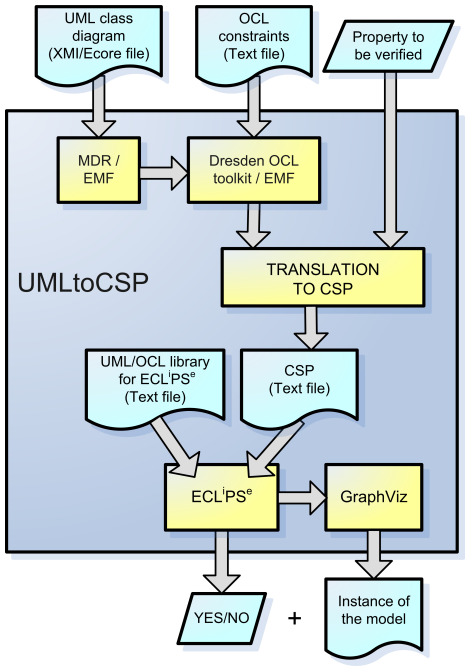


Figure 11: Architecture of UMLtoCSP.

external libraries and tools: the OCL parser from the Dresden OCL toolkit [Dem04], the MDR library for importing XMI files, the EMF (Eclipse Modeling Framework) libraries for importing ECore files, the ECL^{i}PS^e [The07] constraint programming system for solving the CSPs and the GraphViz [Gra] graph visualization package for presenting the results graphically. Figure 11 presents the architecture of the tool.

As shown in the Figure 11, as a first step, the tool permits to import both the UML model and the OCL constraints from an XMI/ECore file and a text file, respectively. More specifically, the UML class diagram can be imported from an XMI (XML Metadata Interchange) file, such as those generated by CASE tools like ArgoUML, or in the ECore format from the Eclipse Modeling Framework[9]. The text file containing the OCL expressions is parsed using the Dresden OCL toolkit. Next, users can choose to generate the CSP from the UML/OCL model. As part of the translation process, users must indicate the correctness property they want to check on the model and (optionally) the ranges for the domains to be used for the variables in the CSP (otherwise default values are used). Finally, the tool generates the CSP with the support of our UML/OCL CSP library (included with the tool). The resulting CSP is executed

[9]EMF refers to the Eclipse IDE (`http://www.eclipse.org`.) not the ECL^{i}PS^e constraint solver.

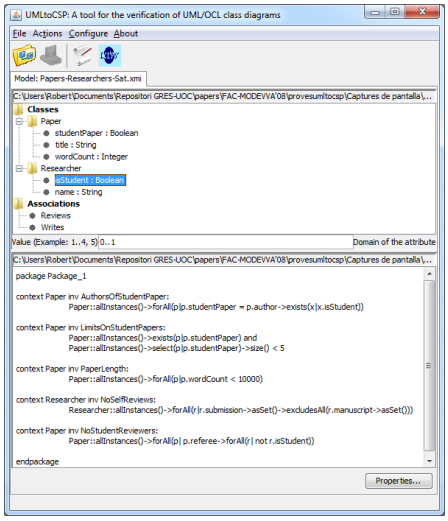
using the ECL^{i}PS^e constraint solver API to try to find one solution for it, and thus, to determine the correctness of the original UML/OCL model. When such a solution exists, it is shown to the user graphically as an object diagram. If no solution exists, the tool prompts a message explaining this fact and suggesting a revision or the model or a change in the size of the search state space.

Figure 12 illustrates the graphical user interface used in this verification flow. Figure 12(a) shows the initial screen once the input models have been parsed. The classes and associations in the diagram are displayed in a tree view, and the textual OCL constraints appear in a separate frame. Also in this window, designers may optionally parametrize the search space by defining the domain of each attribute, the number of objects of a class or the number of links in an association. If the user does not feel the need to customize the search space, the tool automatically uses the suggested default domains based on the type of the attribute, e.g. 0 or 1 for boolean attributes and a finite range for integer attributes. For example, Figure 12 (a) shows how the user assigns the domain of the boolean attribute *isStudent*, which is 0..1 by default. To modify the domain, the user selects the attribute and writes the new domain in the text field "Value" below. A domain can be defined as a single value, an interval of values or a list of values and intervals.

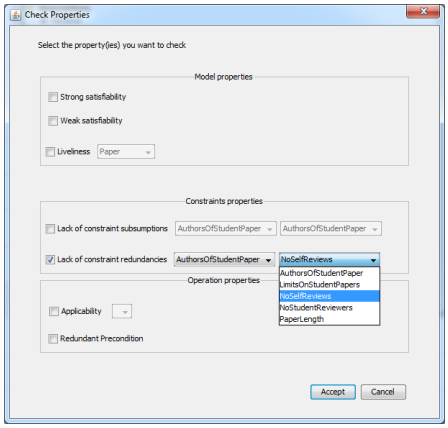
Then, using the menu from Fig. 12 (b), designers can select the properties of interest for the analysis. For properties like liveliness or constraint redundancy which affect a specific class or constraint, a drop-down list provides the list of candidates from the model. After this step, verification is fully automated: the tool translates the UML class diagram, OCL constraints and the correctness properties into a CSP, which is passed to the ECL^{i}PS^e constraint solver API to find a solution to the CSP, and thus, to determine the correctness of the original UML/OCL model. When such a solution exists, it is shown to the user graphically as an object diagram. Fig. 12 (c) shows an example of a result depicted by UMLtoCSP. In particular, it considers a modified version of the running example from Figure 1 where the multiplicities of the roles *submission* and *manuscript* have been changed to 0..1, i.e. there can be authors that do not review papers and reviewers that do not write papers. This modified version is strongly satisfiable, as shown by the object diagram displayed by the tool which satisfies all the constraints of the model: papers have one or two authors and three referees, no author reviews his own paper, the maximum paper length is not exceeded, student papers have at least one student author, students do not act as reviewers and there are between 1 and 5 student papers.

An advantage of this architecture is that the translation and verification are completely hidden from the designer. Therefore, users of UMLtoCSP do not need any kind of knowledge of constraint programming to analyze a model and interpret the results, since the results provided by the CSP solver are reinterpreted in terms of the original UML/OCL models and returned to the user as a UML object diagram that the user can directly understand. Furthermore, they can provide the input to the tool directly as a UML class diagram in the format being used by their CASE tool.

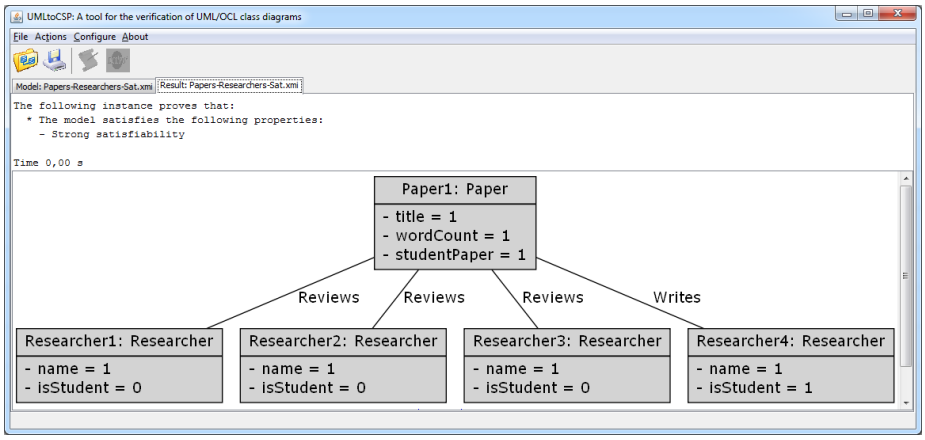
As future work, we are exploring the full integration of UMLtoCSP as a plug-



(a)



(b)



(c)

Figure 12: GUI of the UMLtoCSP tool: (a) loading the model, (b) selecting the property to be verified, (c) showing the result of the analysis on a modified version of the running example.

in of the Eclipse IDE, in order to further improve its usability and facilitate its use by the Eclipse community.

8. Problem Complexity and Efficiency Issues

This section discusses the complexity of the problem we are dealing with and the strategies we have followed to improve the efficiency of our method.

8.1. Problem Size and Complexity

Reasoning on UML class diagrams is EXPTIME-complete [BCG05] and, when general OCL constraints are allowed, it becomes undecidable. Therefore, if this problem is addressed as a search problem, i.e. locating a correct or incorrect instance within the space of all possible instances, a careful analysis of this search space is required. The goal of this preliminary analysis is extracting useful heuristics that can guide the search in order to make it more efficient. Also, to ensure termination, search will focus on a finite subset of the potentially infinite search space. Thus, it is necessary to precisely define the bounds for this subset.

The search space can be organized as a *search tree*, where each leaf corresponds to a solution (either feasible or unfeasible) and each internal node represents a decision in the search process (assigning a value to a variable from the domain of eligible values). The difficulty of a search problem depends on a variety of factors, such as the *size* of the search tree, the number of *feasible* solutions it contains and the effectiveness of the search optimizations (see section 8.2.1). Given that the last factors are problem-dependent, we will focus our discussion on search tree size, as it can provide an intuition of the upper bound on the number of solutions to be explored and the scalability issues for problems with similar constraints.

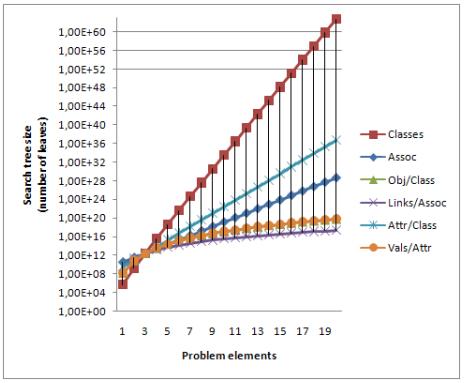
In general, the size of search tree is determined as the product of all the cardinalities of the domains for each variable in the problem. In order to calculate it, for the specific problem we are working in, variables and domains introduced in section 4 are used.

- For the classes: we consider the number of possible objects per class c ($|domain(size_c)|$) and, for each class, the number of possible values of each attribute f_i ($|domain(f_i)|$).

$$\prod_{c \in Cl} |domain(size_c)| \cdot \prod_{c \in Cl} size_c \cdot (size_c \cdot \prod_{f_i \in f} |domain(f_i)|) \qquad (1)$$

- For the associations: we consider the number of links in each association as ($|domain(size_{as})|$) and, in each association, the number of objects in each participating class p_i ($|domain(p_i)|$).

$$\prod_{as \in As} |domain(size_{as})| \cdot \prod_{as \in As} size_{as} \cdot \prod_{p_i \in PCas} |domain(p_i)| \qquad (2)$$



Classes Number of classes in the UML design

Assoc Number of associations in the UML design

Obj/Class Number of objects of a given class

Links/Assoc Number of links per association

Attr/Class Number of attributes of a given class

Vals/Attr Number of possible values for a given attribute

Figure 13: Search tree leaves depending on the size of the input model (logarithmic scale).

The total number of leaves can be found multiplying Eq.'s 1 and 2. It is easy to see that, even for small models, it is not possible to visit the whole tree in order to get the solution. Several parameters of the input model will affect the size of this search tree: the number of classes and attributes of the model, the number of attributes per class, the domain of each attribute and the number of allowed objects/links per class/association. In order to provide a rough idea of the magnitude of this size and the effect of these parameters, Fig. 13 illustrates some sample data. The graphic has been built fixing, for each parameter line, the rest of parameters to three. This is an average value and permits to analyse the evolution of the tree search size increasing a single parameter. It is easy to notice that a unitary increase in any of these parameters implies an exponential growth in the size of the search tree.

8.2. Search Strategy

Efficiency improvements can be implemented at run-time level (i.e. parametrising the CSP solver by selecting the traversal algorithm, controlling the search,...) or at design-level (choosing the right set of variables, constraints and domains for the CSP in order to optimize the search).

8.2.1. Back-tracking, Search Tree Pruning and Search Control

In Constraint Programming (CP) the search space defined by variables and their domains is visited by a depth-first search algorithm. Each step of the traversal process assigns a value to a variable, extending the current partial solution until a complete solution is found in a leaf. Although many other approaches have been explored, backtracking is the most commonly used for this traversal.

One of the drawbacks of backtracking is the *late detection of inconsistencies*, i.e. the failure of the search algorithm to find out that the branch currently being explored does not lead to a feasible solution until it reaches the end and evaluates its feasibility. In order to avoid this, constraints among variables are used by

CP to obviate visiting non feasible solutions. This is achieved by propagation (see section 3). Thus, with the help of forward checking [Tsa93] and other consistency techniques (e.g. node-consistency, arc-consistency, etc. [Tsa93]), constraints help the search engine to detect more quickly those branches that do not lead to feasible solutions.

Thanks to these techniques, it is possible to detect that a partial solution cannot possibly be extended into a feasible solution. In this way, backtracks can be performed without having to compute each complete unfeasible solution. Thus, these techniques partially avoid the main disadvantage of backtracking mentioned before: the search engine backtracks when a decision just taken implies no further feasible solutions and hence no need to go on that branch of the tree. These optimizations may greatly reduce the number of visited leaves in the search tree.

However, the benefits of constraint propagation cannot be quantified in general, since they are completely problem-dependent. The $\mathsf{ECL^iPS^e}$ solver already uses both backtracking and constraint propagation techniques by default.

Besides this, and differently from other paradigms, CP allows to control the search phase, in order to speed it up. This means that selecting the most efficient structure and traversal of the search is fundamental, and these are determined by (1) the order in which variables are assigned a value and (2) the order in which potential values are selected. Several heuristics come usually, by default, within CP languages, e.g. assign first the variable with the most constraints or assign first the smallest value within a domain. However, users can program other heuristics useful for their specific problems. There are neither better nor worse heuristics: their adequacy is dependent on the problem so it is important to check all the combinations and analyse the reasons making one better than others.

During the tool construction, all the possible combinations of general-purpose heuristics offered by the $\mathsf{ECL^iPS^e}$ solver for variable ordering and value selection have been evaluated in several examples. Studying the results, we have selected suitable search heuristics for the structural and global subproblems. For example:

- In the structural problem, in most cases it makes sense to start assigning first the smallest values to class/association sizes. This makes the solver check small instances first, getting smaller counterexamples and improving the likelihood of getting a faster response, as small instances can be checked faster.

- In the global subproblem, the choice of values for attributes is completely problem dependent, so the fastest and simplest heuristic (pick the first value available in the domain) is used.

This choice of heuristics may not be optimal for all problems, but it is a conservative selection which is likely to work well in general and relieves users from the burden of having to choose an heuristic for themselves.

Further Search Improvements. Apart from deciding the way the search tree is visited, some specific features of a problem could lead the programmer to make additional decisions. Typical improvements are the splitting of the set of variables into independent subsets (or like in our case, into directly dependent groups), the removal of both non interesting variables and structural search tree symmetries, etc. These improvements are left for further work.

8.2.2. Search Design

In what follows, we present the optimization strategies we follow during the generation of the CSP.

UML/OCL Search: dependent variable subsets. Given the CSP translated from a model, it is easy to detect two kinds of directly dependent variables: on the one hand, those indicating the number of instances of a class or association (i.e. the objects and the links) and, on the other hand, those representing everything contained by the objects and links themselves (i.e. oids, attributes, etc.). It is clear that the latter depend directly on the instantiation of the former ones. Thus, the objects of a class cannot be created until the number of objects of that class is already known[10]. This approach creates two dependent subsets of variables: when a variable in the first subset is bound, its corresponding set of variables in the second one can be created and instantiated. Surprisingly enough, the first subset corresponds to those variables modeling the UML schema, while the second models the OCL constraints.

This peculiarity takes us to divide the search into two steps we call UML and OCL steps — or structural constraints and global constraints problems (see Fig. 10). In a first stage, variables related to the UML class diagram are bound, and after finding a complete instantiation for this first subproblem, objects and associations are created. From this point, a second search is performed for the second subproblem. The two main reasons for doing this are:

- The dependence between both subsets of variables complicates the management of the corresponding variables. The splitting of the search reduces this complexity.
- Often, the problem to solve does not contain OCL constraints and hence only the first part of the tree is generated and after the creation of objects and associations, these are automatically instantiated — with no backtracking at all.

The following two sections analyse the design of these two stages.

UML Search: Structural Subproblem. The first CSP — the structural subproblem — is defined by:

1. Variables for the number of objects of each class, $Size_c$.

[10]There are alternative ways to model this scenario.

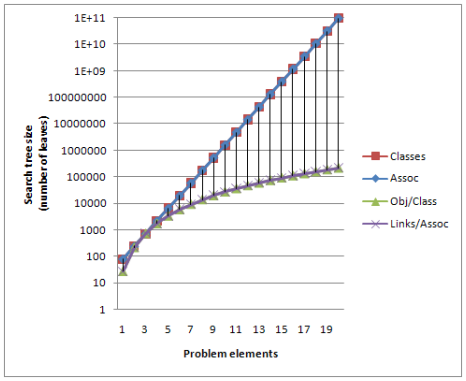


Figure 14: UML search tree size for different problem sizes

2. Variables for the number of links of each association, $Size_{as}$.
3. Constraints related to the cardinalities of associations (see section 4): *bounds on cardinalities* and *number of links*.
4. Constraints removing symmetries (see below).

The size of the tree search for this subproblem is given by Eq.3.

$$\prod_{c \in Cl} |domain(size_c)| \cdot \prod_{as \in As} |domain(size_{as})| \tag{3}$$

Fig. 14 shows the evolution of the size of the search tree for this subproblem when changing input parameters. Although this is a small problem compared with the complete one, it is still EXPTIME-complete.

OCL Search: Global Constraints. The second CSP — the global constraints subproblem — is defined by:

1. Variables for the objects of each class, $Instances_c$.
2. Variables for the links of each association, $Instances_{as}$.
3. Constraints related to the OCL rules of the model: *number of instances*, *distinct oids*, *uniqueness of links*, etc.
4. Constraints removing symmetries (see below).

The size of the tree search for this subproblem is given by Eq. 4. Notice that all but two parameters ($Instances_c$ and $Instances_{as}$) have been bound to an integer value in the previous stage. Furthermore, it is important to take into account that there is one of these search trees for every leaf of the previous stage, i.e. the solution found for the structural subproblem.

$$\prod_{c \in Cl} size_c \cdot (size_c \cdot \prod_{f_i \in f} |domain(f_i)|) \cdot \prod_{as \in As} size_{as} \cdot \prod_{p_i \in PCas} |domain(p_i)| \tag{4}$$

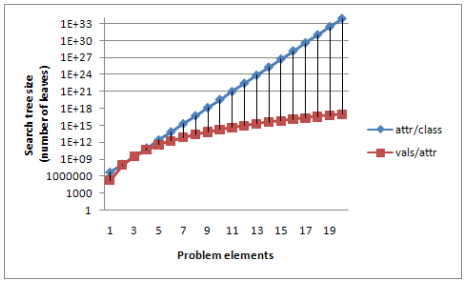


Figure 15: OCL search tree size for different problem sizes

Fig. 15 shows the evolution of the size of the search tree for this subproblem when changing input parameters.

To improve the efficiency of this second problem, we have implemented the next two strategies.

Symmetry removal. The encoding used to map a problem as a CSP should try to avoid potential *symmetries*, i.e. the fact that several assignments to variables of the CSP are equivalent because they correspond to a single solution of the original problem. Symmetries are undesirable because they cause additional overhead to the solver: it has to waste time checking the same solution several times, once for every symmetric assignment.

Symmetry removal is the process of ensuring that the solver considers only one assignment for each family of symmetric solutions. There are several techniques for removing symmetries from a CSP encoding. One of the most effective techniques is the inclusion of additional constraints in the CSP which forbid alternative symmetric assignments. For instance, a typical symmetry removal constraint is requiring part of the assignment to be sorted according to some ordering criteria. This can be done, for instance, when the order among different solutions does not matter: the solver will only consider the smallest solution according to the ordering, discarding the rest.

The choice of these additional constraints depends on the specific CSP encoding used for the problem. Notice however that symmetry removal constraints also cause an overhead to the solver, as they need to be evaluated during the search. For practical reasons, symmetry removal constraints should not be overly expensive to evaluate: the solver should execute faster with these symmetry removal constraints than without them. Therefore, symmetries which are very complex to detect will not be removed.

In the context of UML/OCL diagrams, there are several degrees of symmetry. First, each instance of the diagram can be abstracted as a labeled graph, where objects are the vertices, associations are the edges and each object is labeled with its type and attribute values. Intuitively, if among two instances there is a graph isomorphism that preserves labels, it means that both instances are equivalent. However, detecting graph isomorphism is computationally complex and thus trying to avoid this symmetry would be counterproductive. In-

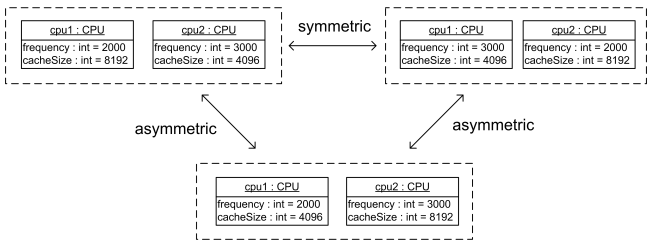


Figure 16: Example of symmetries in UML/OCL diagrams.

stead, we will focus on another type of symmetries caused by our encoding of instances into variables: the ordering of instances of a class or association. In our encoding, variables are assigned sequentially and therefore there is an implicit ordering among them. However, from the point of view of the instance, the ordering of objects and links is irrelevant. For instance, if there are two objects of the same class, there will be two possible assignments to their attributes which will be symmetric. That is, they all can be swapped between these two objects without changing the solution. Figure 16 illustrates this example for a class diagram with a single class (*CPU*) and two attributes (*frequency* and *cacheSize*). The two instances at the top are symmetric, while they are distinct from the one at the bottom. This symmetry can be removed by adding a constraint enforcing the attributes of the objects of class CPU to be ordered lexicographically according to the pair (*frequency*, *cacheSize*). In this way, the instance on the top right would be discarded by the solver because the constraint $(3000, 4096) \leq (2000, 8192)$ does not hold. Similarly, it is possible to remove symmetries among links of an association by imposing a lexicographic ordering among links.

Ruling out irrelevant attributes and associations. Reducing the number of variables and/or simplifying the structure of the CSP has also a clear impact in search time. Reducing the number of variables reduces the height of the search tree (since every variable makes the search tree one level deeper). Therefore, during the translation process we avoid translating those model elements that do not affect the verification process. In particular:

- We discard all attributes that do not participate in any of the constraints in the model. A correct instantiation may contain any value in those attributes.
- We discard associations that are not referenced (i.e. navigated) in any constraint and that do not state any multiplicity constraint (all participants have a "0..*" multiplicity). The population of those associations does not affect the correctness of a solution of the CSP.

The next section describes the efficiency level achieved by our tool once all the optimizations described herein have been integrated.

Name	#Class	#Assoc	#Attrib	#Inv	#Obj	#Links	CPU
running-unsat	2	2	5	5	–	–	0.00s
running-sat	2	2	5	5	5	4	0.00s
eShop-unsat	7	4	9	2	–	–	0.01s
eShop-sat	7	4	9	2	8	7	0.16s
qvt-applic	11	5	7	8	7	6	0.02s
tgg-total	11	5	7	8	14	17	2.41s
prod-sys-cm	15	6	2	4	5	4	0.83s
prod-sys-wc	15	6	2	2	5	3	1.11s

Table 1: Experimental results using UMLtoCSP.

8.3. Experimental Results and Efficiency Assessment

Table 1 describes experimental results for the verification of strong satisfiability using UMLtoCSP. The examples include the running example used in the paper, a model of an e-commerce site, a model transformation specification using TGG or QVT and a model of the production system in a factory. For each example, the number of classes, associations, attributes and OCL invariants is provided as well. This table reports the execution time of the verification process measured on a Intel Core 2 Duo 2.53 Ghz with 2 Gb RAM. Only for those examples that are satisfiable, we provide the number of objects and links of the satisfying instance constructed by UMLtoCSP.

The following state space bounds have been used in the verification of these examples: 2 objects per class, 3 values per integer attribute and 3 links per association. Given the combinatorial explosion of the search problem, using larger domains would result in longer execution times. For example, increasing the search space size up to 5 objects per class, 10 integer values per attribute and 10 links per association causes the CPU time of the eShop-sat example to raise up to 318.1 seconds, as the solver has to consider instances with up to 35 objects and 40 links. Thus, a rule of thumb when using a bounded verification tool like UMLtoCSP is *incremental scoping*: start using small bounds to get a quick answer and, if the answer is inconclusive, progressively use larger domains.

No experimental data is provided for unsatisfiable versions of the larger examples in the table. The rationale behind this lack of data is that the original versions of these examples were satisfiable and, thus, the inconsistency had to be artificially introduced. During the experimentation, we detected that the type of unsatisfiability being inserted had a large impact in the results, producing a large variability: from the answer being computed instantly to the search timing out (no answer in 10 minutes). Given that this choice is purely artificial and leads to one-sided results in one direction or the other, rather than providing a specific example we simply state that some unsatisfiable versions of those problems produce a time out.

As a general rule, UMLtoCSP may be most efficient when the underlying problem is satisfiable. The reason is that, like model checkers or SAT solvers, the search stops when the first feasible solution is found, so the rest of state space does not need to be explored. Therefore, UMLtoCSP is best suited as a tool for generating examples, e.g. the automatic generation of test cases,

the computation of counterexamples for a given property or the creation of snapshots for model validation. In any case, there can be exceptions to this rule for diagrams where constraint optimizations is exceptionally effective in pruning paths not leading to feasible solutions.

In order to consider the application of the tool to even larger diagrams, we have performed additional experiments considering three scenarios based on the class diagrams in Figs. 17 and 18. These are artificial examples, where a "ring" of n classes are connected by n binary associations. These scenarios are designed in order to make it scalable for arbitrary values of n, and thus, to provide information on the scalability of the tool.

In the first scenario (Fig. 17), the goal is the analysis of the tool in the structural subproblem. Therefore, we consider that there are no integrity constraints and we focus on the multiplicities of association ends. This class diagram would be strongly satisfiable if all those multiplicities were 1, e.g. by creating a single object of each class and connecting them through the corresponding associations to the two *neighbour* objects. However, if one of the multiplicities were 2, the class diagram would become unsatisfiable. In this way, it is possible to evaluate the behavior of our tool both with a satisfiable and an unsatisfiable version of the class diagram.

In the second and third scenarios (Fig. 18) we are interested in considering a model with OCL constraints. Hence, we consider that all association ends have a multiplicity of 1..1, making the structural subproblem satisfiable. For the OCL subproblem, we define n constraints, each defining a relationship between the value of an attribute in class i and the value of the corresponding object in class $i + 1$. Depending on the relationship operator that we choose ($>$ or $\geq$), the class diagram may be strongly satisfiable or not.

The difference among the second and third scenarios is the location of the inconsistency. The second scenario (Fig. 18 left) assumes that the inconsistency arises due to the incompatibility of two constraints involving *Class1* and *Class2*. In this sense, the incompatibility is localized in a fragment of the class diagram. In contrast, the third scenario considers a case where the incompatibility arises from the interaction of all constraints in the model, which establish a cyclic dependency on the values of the attributes of all classes. Precisely, the unsatisfiable version of this third scenario has been designed as the *worst-case scenario* for our approach, as all variables of a potential solution have to be assigned in order to detect that the solution is unfeasible.

These examples have been tested for models of different sizes, consisting of 2, 5, 10, 100 and 1000 classes on a Xeon 5050 3Ghz with 4Gb of RAM. All these examples have been measured with the following domains: the number of objects per class can be between 0 and 5, there are three possible values for each attribute and the number of links in each association is between 0 and 10. The reported execution times consider only the verification of the model, excluding the time required to parse the input XMI and OCL files.

Table 2 details the experimental results for these examples, studying both the satisfiable and unsatisfiable versions for each model size. From these results, it can be inferred that depending on the structure of the diagram and the con-

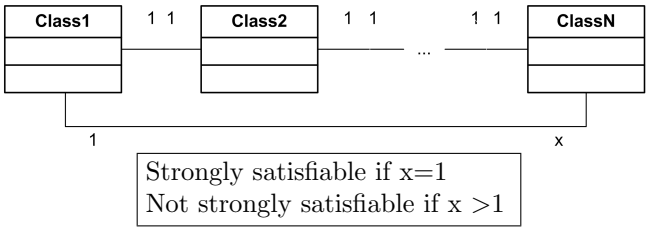


Figure 17: Example without OCL constraints.

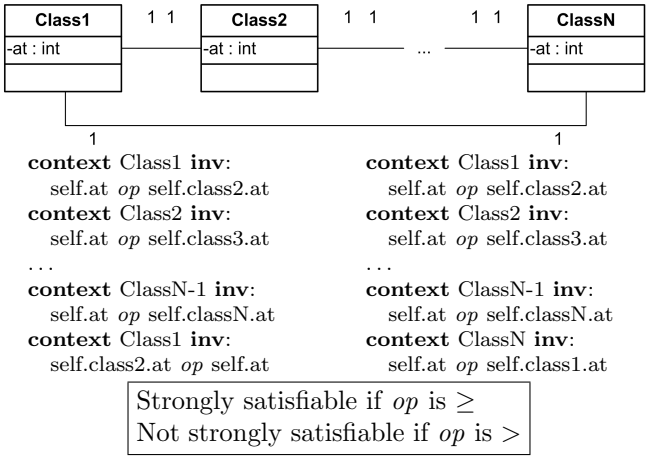


Figure 18: Examples with OCL constraints: inconsistency in a model fragment (left) or in the entire model (right).

	Figure 17 (no OCL)		**Figure 18 (with OCL invariants)**			
			Model fragment (left)		**Entire model (right)**	
n	**Sat**	**Unsat**	**Sat**	**Unsat**	**Sat**	**Unsat**
2	0.00s	0.00s	0.00s	3.67s	0.00s	1.56s
5	0.01s	0.00s	0.01s	3.78s	0.01s	1.84s
10	0.01s	0.00s	0.01s	3.78s	0.01s	5146.70s
100	0.14s	0.05s	0.17s	4.50s	0.17s	—
1000	3.58s	2.07s	3.81s	31.55s	4.97s	—

Table 2: Execution time for $n = 2, 5, 10, 100$ and 1000 classes, comparing the performance in satisfiable (Sat) and non-satisfiable (Unsat) problems.

Tool	Formalism	T	V	Limitations
[BCG05, SMSJ03, BM08]	Description Logics	A	A	No OCL support
[CCGM04, MM06]	CSP	M	A	No OCL support, bounded verification
[MB07]	Linear Programming	A	A	No OCL support
Alloy [Jac06]	Relational Logics	M/A	A	Bounded verification, limited arithmetic support
[SWK+10]	SAT	A	A	Bounded verification, limited arithmetic support
[WBBK10]	Syntax patterns	A	A	Incomplete, limited to specific constraint patterns
HOL-OCL [BW09]	Higher-Order Logics	A	U	Undecidability
PVS [KFdB+05]	Higher-Order Logics	A	U	Undecidability
[QRT+10, QT12]	Deductive DB queries	A	A	Limited arithmetic support
USE [GBR05]	ASSL	M	A	Validation only
[AIAB11]	Genetic Algorithm	A	A	Incomplete
UMLtoCSP	CSP	A	A	Bounded verification, Limitations in section 4.4

Legend: T Translation, V Verification, A automatic, M Manual, U user-assisted.

Table 3: Comparison of several methods for the verification of UML/OCL class diagrams.

straints it contains, it may be possible to analyze large class diagrams efficiently using UMLtoCSP. However, some diagrams may face scalability problems with as little as 10 classes, as illustrated by the worst-case scenario in the last column.

9. Related work

In this section, we will compare our approach with the related work in the area of static consistency analysis of class diagrams. We will not discuss extensions of this work to deal with dynamic properties, e.g. model checking or analysis of operations contracts e.g. [Jac06, QT09, CCR09, BHS07]. Furthermore, we will restrict ourselves to the application of consistency analysis to model verification and validation. Even though the examples and counterexamples computed by these tools can also be applied to test-case generation [DTGF06, WS08], research on model-based testing focuses on a more abstract problem, the definition of suitable testing criteria, while the generation of tests cases for a given testing criterion is solved using the tools described in this section.

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. As it was mentioned before, reasoning on UML class diagrams is EXPTIME-complete without OCL constraints and undecidable when general OCL constraints are allowed. By choosing a particular formalism, each method commits to a different trade-off regarding the verification of correctness properties of UML/OCL diagrams. Table 3 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 supports a rich family of OCL constraints (the limitations on the UML/OCL subset supported by UMLtoCSP are described in section 4.4). Additionally, our tool is able to provide valid instantiations for satisfiable models.

Several previous works focus on the verification of UML class diagrams *without* OCL constraints (or just with some specific types of basic constraints), i.e. the decidable version of the problem. Some approaches in this category are based, among others, on Description Logics [BCG05, SMSJ03, BM08] or Constraint or Linear Programming [CCGM04, MM06, MB07].

It is also possible to achieve an efficient[11] analysis if only specific patterns of OCL constraints are allowed. Some examples of these *constraint patterns* are the uniqueness of an identifier or the lack of cyclic dependencies among objects. In such cases, it is possible to derive a priori the consistency lemmas required for the constraint pattern to hold. These lemmas can be checked efficiently by finding syntactical patterns in the OCL constraints [WBBK10]. An advantage of this approach is its efficiency, as it is polynomial in the size of the model contrary to the rest of methods dealing with OCL, which have an exponential worst-case behavior. On the other hand, the method is restricted to the analysis of a specific set of constraint patterns and therefore it does not support general OCL constraints. Furthermore, the method is incomplete as the analysis of syntactical patterns may be insufficient to prove the consistency lemmas, even if they hold.

Another related approach is the USE tool [GBR05]. However, USE is more focused on validation than in verification, that is, it permits to construct finite snapshots of a UML model that satisfy a set of OCL constraints but the generation of snapshots is not supposed to be exhaustive: USE does not attempt to automatically explore a whole range of values to determine the correctness of the model. Rather, the generation process is user-driven. Users define a list of desired characteristics for the instances to be created and their number. Then, the tool generates and tests the validity of such instance set. In contrast, our approach is fully automatic. To the best of our knowledge, the approach presented in this paper is the first method addressing the verification of UML class diagrams *with OCL constraints* based on Constraint Programming.

Regarding verification of UML class diagrams *with* general OCL constraints, some examples of formalisms used in this problem are Relational Logics (Alloy [Jac06]), Higher-Order Logics (HOL-OCL [BW06, BW09, KFdB+05]) and deductive database queries (AuRUS [QRT+10, QT12]). However not all these formalisms are as expressive as the UMLtoCSP method. Some approaches support only a subset of UML or OCL constructs, e.g. Alloy has scalability problems in operations involving integers and AuRUS supports only comparisons

[11] Though it is difficult to discuss efficiency of UML verification methods since many approaches have not published efficiency results achieved with them.

while UMLtoCSP supports arithmetic expressions and comparisons. Undecidability also imposes several limitations on some approaches, e.g. requiring user-interaction to complete proofs as in HOL-OCL and PVS. In some cases, it is possible to detect that the analysis of a model will be decidable and improve the efficiency of the verification for that particular model [QT08].

Other works like [AIAB11] consider the problem of *model-based testing*: generating test cases for a software system from its UML/OCL model. Computing test cases, i.e. satisfying instances, is considered a search problem which is solved using a genetic algorithm backed by specially crafted heuristics to measure how far an instance is from satisfying an OCL invariant. Heuristic search can lead to very efficient computations. However, a limitation of this approach is that it is unable to diagnose inconsistent constraints: as the search is not complete, it is not possible to draw conclusions from empty responses.

Among all these approaches, the most similar in terms of features are UML2Alloy [ABGR07] and [SWK$^+$10]. In both approaches, the underlying reasoning engine is a SAT solver: the problem and the correctness property are translated into a boolean formula whose satisfiability needs to be determined. In the case of UML2Alloy, there is an intermediate step, the transformation of the UML/OCL model into the Alloy notation, which the Alloy Analyzer internally translates into a SAT instance. Meanwhile, [SWK$^+$10] proceeds by directly generating the SAT instance and passing it to the SAT solver MiniSAT. UMLtoCSP offers an advantage with respect to these two bounded verification approaches. In SAT-based methods, constraints involving numbers must also be expressed in terms of boolean variables, meaning that (1) users must specify the number of bits being used to encode each value and (2) operations on numbers (e.g. addition, difference, multiplication, 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. In a CSP, increasing the range of a numeric value also increases the search space, but encoding complex arithmetic expressions on integers is straightforward. As an example of this problem, let us consider the running example from Fig. 1. When this model is written in the Alloy notation, we realize that one of the constraints (*PaperLength*) contains the integer constant 10000. Encoding this constant at the boolean level requires 15 bits per integer which requires a large amount of CPU time just to generate the SAT instance (the generation of the SAT instance did not finish after 1 hour of CPU time in an Intel Core Duo T2400 1.83Ghz with 1 Gb RAM), while UMLtoCSP computes the result in less than one second. Therefore, any model where arithmetic constraints are necessary and they may involve large values would be a good candidate to be analyzed with UMLtoCSP.

Another benefit of UMLtoCSP with respect to Alloy is an advantage in terms of usability. UML2Alloy and Alloy are separate tools, meaning that a user has to launch UML2Alloy, load the model and translate it, then launch Alloy, load the translation and verify it. Meanwhile, UMLtoCSP offers an integrated environment for verification which also supports the Ecore format, and therefore, the range of Eclipse EMF tools. On the other hand, it should be noted

that Alloy is a mature tool, with a consolidated implementation. For example, some interesting features of Alloy which are not supported by UMLtoCSP are bounded model checking capabilities and the computation of *unsatisfiable cores* [TCJ08], i.e. minimal sets of conflicting constraints within the model.

Furthermore, our approach does not impose theoretical limitations that restrict any UML or OCL constructs besides those identified in section 4.4. Though a formal proof is outside the scope of this paper, we argue that our approach terminates for any input but it is *not complete*: results are only conclusive if a feasible solution to the CSP is found. In that sense, our method only guarantees that if a solution to the CSP exists within the parameters 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. An observation which alleviates this limitation is the *small scope hypothesis* [Jac06], i.e. it is possible to identify a large percentage of errors in a system by considering all possible instances within small domain. This limitation is shared by Alloy, while HOL-COL and the AuRUS method provide complete proof procedures.

Nonetheless, an efficient decidable procedure may provide useful information even if the answer is not conclusive. For example, models which do not have any correct instances with a small population may require a closer inspection.

10. 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 the prototype tool UMLtoCSP.

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 with respect to alternative approaches. However, if desired, it is also possible to use the transformation of UML/OCL models into CSPs on infinite domains: constraint solvers also allow an *incomplete search* [AW07] although termination is not guaranteed and depends on heuristics to guide the search process. In that way, our method would become semidecidable but complete (for properties that can be satisfied by *finite* instances). Moreover, the instance generation nature of this approach (i.e. the fact that properties are proven by creating legal instances of the model), makes it amenable for model validation or test generation purposes.

As a further work we would like to refine our translation process to improve the efficiency of the obtained CSP. In particular we would like to advance in the automatic definition of appropriate ranges for attribute domains (based on the semantics of the OCL constraints that reference them), in the selection of the best constraint programming search strategies for this particular class of generated CSPs and in the extraction of basic (implicit) constraints from complex

OCL constraints that can be used by the solver to improve constraint propagation [YBP07]. We consider that the paradigm of *abstract interpretation* [CC77] would be well-suited for performing this static analysis of OCL invariants.

Regarding the application of this technique to related problems, preliminary results on the verification of correctness properties of UML/OCL models annotated with declarative operation contracts (pre-condition/post-condition) have been described in [CCR09]. Our goal is to extend these results, e.g. to support reasoning on sequences of operations. Furthermore, we plan to apply this tool for the analysis of models and meta-models of Domain-Specific Languages (DSLs), in order to detect potential inconsistencies in the definition of new DSLs.

Acknowledgements

This work is partially funded by the Spanish Ministry of Science and Innovation through the project "Design and construction of a Conceptual Modeling Assistant" (TIN2008-00444/TIN - Grupo Consolidado) and a research grant from the Internet Interdisciplinary Institute (IN3) at UOC. The authors would like to thank Patricia de la Fuente, Christian Pérez and David Gañán for their work on the implementation of the tool. The authors would also like to thank the members of the Conceptual Modeling Group (GMC) at the Universitat Politècnica de Catalunya for their comments.

[ABGR07] Kyriakos Anastasakis, Behzad Bordbar, Geri Georg, and Indrakshi Ray. UML2Alloy: A challenging model transformation. In *ACM/IEEE 10th International Conference on Model Driven Engineering Languages and Systems*, pages 436–450, 2007.

[ACIG10] Alessandro Artale, Diego Calvanese, and Angélica Ibánez-García. Full satisfiability of UML class diagrams. In Jeffrey Parsons, Motoshi Saeki, Peretz Shoval, Carson Woo, and Yair Wand, editors, *Proc. of Conceptual Modeling (ER'2010)*, volume 6412 of *Lecture Notes in Computer Science*, pages 317–331. Springer Berlin / Heidelberg, 2010. 10.1007/978-3-642-16373-9-23.

[AIAB11] Shaukat Ali, Muhammad Zohaib Z. Iqbal, Andrea Arcuri, and Lionel C. Briand. A search-based OCL constraint solver for model-based test data generation. In *Proceedings of the 11th International Conference on Quality Software (QSIC'2011)*, pages 41–50. IEEE Computer Society, 2011.

[AW07] Krzysztof R. Apt and Mark G. Wallace. *Constraint Logic Programming using* ECL$^{\text{i}}$PS$^{\text{e}}$. Cambridge University Press, Cambridge, UK, 2007.

[BCG05] Daniela Berardi, Diego Calvanese, and Giuseppe De Giacomo. Reasoning on UML class diagrams. *Artificial Intelligence*, 168:70–118, 2005.

[Ber02] Daniel Berry. Formal methods: the very idea. Some thoughts about why they work when they work. *Science of Computer Programming*, 42(1):11–27, 2002.

[BHS07] Bernhard Beckert, Reiner Hähnle, and Peter H. Schmitt, editors. *Verification of Object-Oriented Software: The KeY Approach*, volume 4334 of *Lecture Notes in Computer Science*. Springer-Verlag, 2007.

[BKW10] Achim Brucker, Matthias Krieger, and Burkhart Wolff. Extending OCL with null-references. In *Models in Software Engineering*, volume 6002 of *Lecture Notes in Computer Science*, pages 261–275, 2010.

[BM08] Mira Balaban and Azzam Maraee. A UML-based method for deciding finite satisfiability in Description Logics. In *Proc. of the 21st International Workshop on Description Logics (DL'2008)*, volume 353 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2008.

[BW06] Achim D. Brucker and Burkhart Wolff. The HOL-OCL book. Technical Report 525, ETH Zurich, 2006.

[BW09] Achim Brucker and Burkhart Wolff. Semantics, calculi, and analysis for object-oriented specifications. *Acta Informatica*, 46:255–284, 2009. 10.1007/s00236-009-0093-8.

[CC77] Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proc. of the 4th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 238–252. ACM Press, 1977.

[CCGdL10a] Jordi Cabot, Robert Clarisó, Esther Guerra, and Juan de Lara. A uml/ocl framework for the analysis of graph transformation rules. *Software and System Modeling*, 9(3):335–357, 2010.

[CCGdL10b] Jordi Cabot, Robert Clarisó, Esther Guerra, and Juan de Lara. Verification and validation of declarative model-to-model transformations through invariants. *Journal of Systems and Software*, 83(2):283–302, 2010.

[CCGM04] Marco Cadoli, Diego Calvanese, Giuseppe De Giacomo, and Toni 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.

[CCR08] Jordi Cabot, Robert Clarisó, and Daniel Riera. Verification of UML/OCL class diagrams using constraint programming. In *Model Driven Engineering, Verification, and Validation Workshop (MoDeVVa'2008)*, pages 73–80, 2008.

[CCR09] Jordi Cabot, Robert Clarisó, and Daniel Riera. Verifying UML/OCL operation contracts. In *Proc. 7th Int. Conf. on Integrated Formal Methods (IFM'2009)*, volume 5423 of *Lecture Notes in Computer Science*, pages 40–55. Springer-Verlag, 2009.

[Cha76] Peter Pin-Shan Chan. The entity-relationship modeltoward a unified view of data. *ACM Transactions on Database Systems (TODS)*, 1(1):9–36, 1976.

[CHI] CHIP V5. `http://www.cosytec.com/production\_scheduling/chip/optimization\_product\_chip.htm`.

[Com] Comet. `http://dynadec.com/`.

[Cre] Cream. `http://bach.istc.kobe-u.ac.jp/cream/`.

[CT06] Jordi Cabot and Ernest Teniente. Constraint support in MDA tools: A survey. In Arend Rensink and Jos Warmer, editors, *Model Driven Architecture: Foundations and Applications*, volume 4066 of *Lecture Notes in Computer Science*, pages 256–267. Springer Berlin Heidelberg, 2006.

[Dem04] Birgit 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.

[DTGF06] Trung T. Dinh-Trong, Sudipto Ghosh, and Robert B. France. A systematic approach to generate inputs to test UML design models. In *17th International Symposium on Software Reliability Engineering (ISSRE'2006)*, pages 95–104. IEEE Computer Society, 2006.

[ER03] Albert Endes and Dieter Rombach. *A Handbook of Software and Systems Engineering: Empirical Observations, Laws and Theories*. Fraunhofer IESE Series on Software Engineering. Addison-Wesley, 2003.

[GBR05] Martin Gogolla, J orn Bohling, and Mark Richters. Validating UML and OCL models in USE by automatic snapshot generation. *Journal on Software and System Modeling*, 4(4):386–398, 2005.

[GNU] GNU-Prolog. `http://www.gprolog.org/`.

[GR02] Martin Gogolla and Mark Richters. Expressing UML class diagrams properties with OCL. In A. Clark and J. Warmer, editors, *Object Modeling with the OCL, The Rationale behind the Object Constraint Language*, pages 85–114, London, UK, 2002. Springer-Verlag.

[Gra] Graphviz. Graph visualization software. `http://www.graphviz.org`.

[HWD08] Florian Heidenreich, Christian Wende, and Birgit Demuth. A framework for generating query language code from ocl invariants. *ECEASST*, 9, 2008.

[ILO] ILOG CP. `http://www-01.ibm.com/software/integration/optimization/cplex-cp-optimizer/`.

[JaC] JaCoP. `http://jacop.osolpro.com/`.

[Jac06] Daniel Jackson. *Software Abstractions: Logic, Language, and Analysis*. The MIT Press, 2006.

[KFdB+05] Marcel Kyas, Harald Fecher, Frank S. de Boer, Joost Jacob, Jozef Hooman, Mark van der Zwaag, Tamarah Arons, and Hillel Kugler. Formalizing UML models and OCL constraints in PVS. *Electron. Notes Theor. Comput. Sci.*, 115:39–47, January 2005.

[LMT09] Francisco J. Lucas, Fernando Molina, and Ambrosio Toval. A systematic review of UML model consistency management. *Information and Software Technology*, 51(12):1631 – 1645, 2009.

[MB07] Azzam Maraee and Mira Balaban. Efficient reasoning about finite satisfiability of UML class diagrams with constrained generalization sets. In *Proc. of the 3rd Model Driven Architecture- Foundations and Applications (ECMDA-FA'2007)*, volume 4530 of *Lecture Notes in Computer Science*, pages 17–31. Springer-Verlag, 2007.

[MDN09] Parastoo Mohagheghi, Vegard Dehlen, and Tor Neple. Definitions and approaches to model quality in model-based software development - a review of literature. *Information and Software Technology*, 51(12):1646 – 1669, 2009. Quality of UML Models.

[MM06] Hugues Malgouyres and Gilles 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.

[MS98] Kim Marriott and Peter J. Stuckey. *Programming with Constraints: An Introduction*. MIT Press, Cambridge, Massachussetts, 1998.

[Obj10] Object Management Group. *OCL 2.2 Specification*, 2010.

[Obj11] Object Management Group. *UML 2.4.1 Specification*, 2011.

[Oli07] A. Olivé. *Conceptual Modeling of Information Systems*. Springer-Verlag Berlin Heidelberg, 2007.

[Oz] Oz. `http://www.mozart-oz.org/`.

[QRT+10] Anna Queralt, Guillem Rull, Ernest Teniente, Carles Farré, and Toni Urpí. Aurus: Automated reasoning on UML/OCL schemas. In *29th International Conference on Conceptual Modeling (ER'2010)*, pages 438–444, 2010.

[QT08] Anna Queralt and Ernest Teniente. Decidable reasoning in UML schemas with constraints. In Zohra Bellahsene and Michel Léonard, editors, *Proc. of the 20th Int. Conf. on Advanced Information Systems Engineering, (CAiSE'2008)*, volume 5074 of *Lecture Notes in Computer Science*, pages 281–295. Springer-Verlag, 2008.

[QT09] Anna Queralt and Ernest Teniente. Reasoning on UML conceptual schemas with operations. In *Proc. of the 21st Int. Conf. on Advanced Information Systems Engineering (CAiSE'2009)*, volume 5565 of *Lecture Notes in Computer Science*, pages 47–62. Springer-Verlag, 2009.

[QT12] Anna Queralt and Ernest Teniente. Verification and validation of UML conceptual schemas with OCL constraints. *ACM Trans. Softw. Eng. Methodol.*, 21(2):13, 2012.

[SIC] SICStus. `http://sicstus.sics.se/`.

[SMSJ03] Ragnhild Van Der Straeten, Tom Mens, Jocelyn Simmonds, and Viviane Jonckers. Using description logic to maintain consistency between UML models. In *Proc. of UML'03.*, volume 2863 of *LNCS*, pages 326–340. Springer, 2003.

[Sow00] John F. Sowa. *Knowledge Representation: Logical, Philosophical, and Computational Foundation.* Computer Science Series. Brooks/Cole, 2000.

[SWK+10] Mathias Soeken, Robert Wille, Mirco Kuhlmann, Martin Gogolla, and Rolf Drechsler. Verifying UML/OCL models using boolean satisfiability. In *Design, Automation and Test in Europe (DATE'2010)*, pages 1341–1344. IEEE, 2010.

[TCJ08] Emina Torlak, Felix Sheng-Ho Chang, and Daniel Jackson. Finding minimal unsatisfiable cores of declarative specifications. In *Formal Methods 2008, (FM'2008)*, volume 5014 of *Lecture Notes in Computer Science*, pages 326–341. Springer-Verlag, 2008.

[The07] The ECL$^{\text{i}}$PS$^{\text{e}}$ Constraint Programming System. `http://www.eclipseclp.org`, mar 2007. version 5.10.

[Tsa93] Edward Tsang. *Foundations of Constraint Satisfaction.* Academic Press, 1993.

[U2C] UMLtoCSP. `http://gres.uoc.edu/UMLtoCSP`.

[WBBK10] Michael Wahler, David Basin, Achim D. Brucker, and Jana Koehler. Efficient analysis of pattern-based constraint specifications. *Software and Systems Modeling*, 2010. To appear.

[WS08] Stephan Weißleder and Bernd-Holger Schlingloff. Quality of automatically generated test cases based on ocl expressions. In *International Conference on Software Testing, Verification, and Validation (ICST'2008)*, pages 517–520. IEEE Computer Society, 2008.

[YBP07] Fang Yu, Tevfik Bultan, and Erik Peterson. Automated size analysis for OCL. In *Proc. of the 6th Joint Meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering (ESEC-FSE '07)*, pages 331–340. ACM, 2007.