Verification of UML/OCL Class Diagrams using Constraint Programming

Schnebli Zoltan

December 22, 2019

Contents

Abstract

The aim of this paper is to present a general overview of constraint programming field and present how to verify UML/OCL class diagrams using Constraint Programming.

First we will present a general formulation of the constraint programming problems.Because constraint programming problems vary from task to task, we will present first a brief introduction to the setup for the task. First, we will present a general introduction to the UML/OCL processes with the basic notations. Then we will present what kind of constraints can we extract from these diagrams. After that we will present what definitions will be usefull for defineing the correctness of these models. And at last we will present how to evaluate the generated CSP model.

1 Introduction

1.1 Constraint Programming

In last few years, Constraint Programming has attracted high attention among experts from many areas because of its potential for solving hard real life problems. Not only it is based on a strong theoretical foundation but it is attracting widespread commercial interest as well. Not surprisingly, it has recently been identified by the Association for Computing Machinery as one of the strategic directions in computer research. However, at the same time, Constraint Programming is still one of the least known and understood technologies.

Constraints arise in most areas of human endeavour. They formalise the dependencies in physical worlds and their mathematical abstractions naturally and transparently. A constraint is simply a logical relation among several unknowns (or variables), each taking a value in a given domain. The constraint thus restricts the possible values that variables can take, it represents partial information about the variables of interest. Constraints can also be heterogeneous, so they can bind unknowns from different domains, for example the length (number) with the word (string). The important feature of constraints is their declarative manner, i.e., they specify what relationship must hold without specifying a computational procedure to enforce that relationship. We all use constraints to guide reasoning as a key part of everyday common sense. "I can be there from five to six o'clock", this is a typical constraint we use to plan our time. Naturally, we do not solve one constraint only but a collection of constraints that are rarely independent. This complicates the problem a bit, so, usually, we have to give and take. Constraint programming is the study of computational systems based on constraints. The idea of constraint programming is to solve problems by stating constraints (requirements) about the problem area and, consequently, finding solution satisfying all the constraints. Barták [1999]

1.2 Software verification

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

Unfortunately, formal verification of software models is known to be undecidable in general. This is also the case when focusing on the verification of UML class diagrams extended with OCL constraints: first-order logic (FOL) itself is undecidable in general and OCL is more expressive than FOL. Therefore, to avoid undecidability, existing methods able to reason on UML/OCL diagrams either limit the UML/OCL constructs that may appear in the diagrams, are not automatic or are semi-decidable. Cabot *et al.* [2008]

The aim of this paper is to present a general overview of constraint programming field and present how to verify UML/OCL class diagrams using Constraint Programming.

1.3 Theory

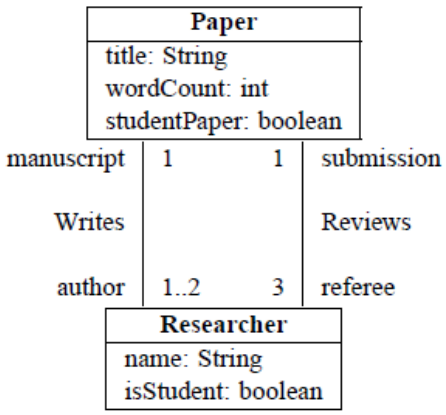
Based on this paper Cabot *et al.* [2008] we will present how using the Constraint Programming paradigm as a complementary method will change the process problem to a fully automatic, decidable and expressive verification of UML/OCL class diagrams.
Decidability is achieved by defining a finite solution space, i.e. establishing finite bounds for the number of instances and finite domains for attribute values to be considered during the verification process. This way, the constraint solver is able to perform a complete search within the solution space.

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

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

1. The multiplicities of association Reviews require exactly three distinct researchers per paper, meanwhile, the multiplicities of Writes requires one or two researchers per paper.

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



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: A UML class diagram with OCL constraints

So in order to detect such unsatisfiability of a model, first, we have to transform the diagram into a CSP. If that CSP has a solution than the model is satisfiable. In the CSP, there are several constraint that restrict the legal values of the variables. To find a solution the constraint solver tries to assign a value to all variables without violating an constraint. If no legal assignment is possible, the model is determined as unsatisfiable.

1.4 Cosntraint programming

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

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

The most traditional technique for finding solutions to such problems is backtracking. These search processes are largely improved by constraint propagation techniques, which means that we are using the information about the structure of the constrainst and the decisions taken so far.

In Barták [1999] they used the ECL$^i PS^e Wallace et al.$ [1997] $a Constraint programming system, where th$ $based language, while the variables may be either simple, structured or lists.$

1.5 Translation of UML/OCL Class Diagrams

Before we can verify the correctness of the diagrams, first, we have to transform them into a CSP. A class diagram is defined as CD = (Cl, As, AC, G, IC), where Cl is the set of

classes, As is the set of associations, AC the set off association classes, G is the set of generalisation sets and IC the set of constraints. Each element is translated into a set of variables, domains and constraints. We are trying to build the smalles domains that suffice to identify inconsistencies.

1.5.1 Tranformation of classes

- A variable $Instances_c$ of type list. Each element represents an instace of c. Therefore the structure: struct(c) = (oid, f_1, ..., f_n), where: oid represents the explicit class identifier and each f_i represents an attribute.

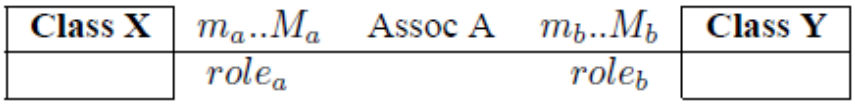
- A variable $Size_c$ of type integer. Its domain is domain($Size_c$) = [0, $PMaxSize_c$], where $PMaxSize_c$ is a parameter which indicates the max number of instances of class c

- Number of instances: $Size_c = length(Instances_c)$

- Distinct oids: $\forall$ x,y $\in Instances_c$: x $\neq$ y $->$ x.oid $\neq$ y.oid

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 domain of the corresponding attribute in c. Boolean and enumerated types are already finite. Finite domains for integers requires at least a lower and an upper bound. For real types we need also a maximum decimal precision and for string types the possible "alphabet" is needed.

The multiplicities of an association impose constraints on the number of instances of the participant classes and the association. These constraints are presented in Fig. 2. First, the set of links is a subset of the cartesian product of the participant classes, so its size (product of class sizes) defines an upper bound for the number of links. Also, minimum and maximum multiplicities of roles define a lower and upper bound relationship between the number of links and the number of objects of each participant class.



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

Figure 2: Implicit cardinality constraints due to the association multiplicities

1.5.2 Definition of correctness properties

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

If the CSP still has a solution once the new constraint is added, we may conclude that the model satisfies the property. The set of correctness properties that can be checked by designers is the following:

Strong satisfiability: The model must have a finite instantiation where the population of all classes and associations is at least one.
Weak satisfiability: The model must have a finite instantiation where the population of at least one class is at least one.
Liveliness of a class c: The model must have a finite instantiation where the population of c is non-empty.
Lack of constraint subsumptions: Given two integrity constraints C1 and C2, the model must have a finite instantiation where C1 is satisfied and C2 is not. Otherwise we can say that C1 subsumes C2 and C2 could be removed.
Lack of constraint redundancies: Given two integrity constraints C1 and C2, the model must have a finite instantiation where only one is satisfied. Otherwise we can say that C1 and C2 are redundant, because for e.g. both have the same truth value and one should be removed.

Other validation criterias can be defined similarly.

1.6 Resolution of the generated CSP

The CSP is organized in two subproblems. In the first one, we define the cardinality variables for the number of instances of each class and association, their domains and all constraints restricting them. In this phase, the goal is to find a legal assignment of values to these variables. If no asssignment is possible, the CSP is directly unfeasible.

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

After each phase we define the variables and theri domains, define the constraints on the variables and finally find a legal assignment to these variables.

2 Conclusion

In this work we presented a general overview of constraint programmingfield and present how to verify UML/OCL class diagrams using Constraint Programming. The general introduction part is followed by a section which contains some recent researches in this domain.

First we presented a general formulation of the constraint programming problems. Because constraint programming problems vary from task to task, we presented first a brief introduction to the setup for the task. First, we presented a general introduction to the UML/OCL processes with the basic notations. Then we presented what kind of constraints can we extract from these diagrams. After that we presented what definitions will be usefull for defineing the correctness of these models. And at last we presented how to evaluate the generated CSP model.

References

Roman Barták. Constraint programming: In pursuit of the holy grail. In *Proceedings of the Week of Doctoral Students (WDS99)*, volume 4, pages 555–564. MatFyzPress Prague, 1999.

Jordi Cabot, Robert Claris, Daniel Riera, et al. Verification of uml/ocl class diagrams using constraint programming. In *2008 IEEE International Conference on Software Testing Verification and Validation Workshop*, pages 73–80. IEEE, 2008.

Mark Wallace, Stefano Novello, and Joachim Schimpf. Eclipse: A platform for constraint logic programming. *ICL Systems Journal*, 12(1):159–200, 1997.