Finite Satisfiability of UML class diagrams by Constraint Programming

Marco Cadoli[1], Diego Calvanese[2], Giuseppe De Giacomo[1], Toni Mancini[1]

[1] Dipartimento di Informatica e Sistemistica
Università di Roma "La Sapienza"
Via Salaria 113, I-00198 Roma, Italy
cadoli|degiacomo|tmancini@dis.uniroma1.it
www.dis.uniroma1.it/~cadoli|~degiacomo|~tmancini
[2] Faculty of Computer Science
Free University of Bolzano/Bozen
Piazza Domenicani 3, I-39100 Bolzano, Italy
calvanese@inf.unibz.it

Abstract. Finite model reasoning in UML class diagrams, e.g., checking whether a class is forced to have either zero or infinitely many objects, is of crucial importance for assessing quality of the analysis phase in software development. Despite the fact that finite model reasoning is often considered more important than unrestricted reasoning, no implementation of the former task has been attempted so far. The main result of this paper is that it is possible to use off-the-shelf tools for constraint modeling and programming for obtaining a finite model reasoner. In particular, exploiting appropriate reasoning techniques, we propose an encoding as a CSP of UML class diagram satisfiability. Moreover, we show also how CP can be used to actually return a finite model of a class diagram. A description of our system, which accepts as input class diagrams in the MOF syntax, and the results of the experimentation performed on the CIM knowledge base are given.

1 Introduction

The Unified Modelling Language (UML, [13], cf. www.uml.org) is probably the most used modelling language in the context of software development, and has been proven to be very effective for the analysis and design phases of the software life cycle.

UML offers a number of diagrams for representing various aspects of the requirements for a software application. Probably the most important diagram is the *class diagram*, which represents all main structural aspects of an application. A typical class diagram shows:

- *classes*, i.e., homogeneous collections of *objects*, i.e., instances;
- *associations*, i.e., relations between classes;
- *ISA hierarchies* between classes, i.e., relations establishing that each object of a class is also an object of another class;

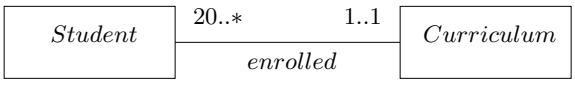

Fig. 1. A UML class diagram.

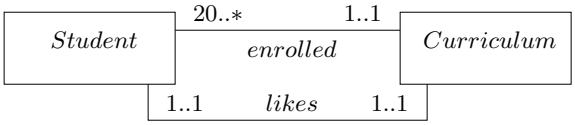

Fig. 2. A UML class diagram with finitely inconsistent classes.

– *multiplicity constraints* on associations, i.e., restrictions on the number of links between objects related by an association.

Actually, a UML class diagram represents also other aspects, e.g., the attributes and the operations of a class, the attributes of an association, and the specialization of an association. Such aspects, for the sake of simplicity, will not be considered in this paper.

An example of a class diagram is shown in Figure 1, which refers to an application concerning management of administrative data of a university, and exhibits two classes (*Student* and *Curriculum*) and an association (*enrolled*) between them. The multiplicity constraints state that:

– Each student must be enrolled in at least one and at most one curriculum;
– Each curriculum must have at least twenty enrolled students, and there is no maximum on the number of enrolled students per curriculum.

It is interesting to note that a class diagram induces restrictions on the number of objects. As an example, referring to the situation of Figure 1, it is possible to have zero, twenty, or more students, but it is impossible to have any number of students between one and nineteen. The reason is that if we had, e.g., five students, then we would need at least one curriculum, which in turn requires at least twenty students.

In some cases the number of objects of a class is forced to be zero. As an example, if we add to the class diagram of Figure 1 a further *likes* association, with the constraints that each student likes exactly one curriculum, and that each curriculum is liked by exactly one student (cf. Figure 2), then it is impossible to have any finite non-zero number of students and curricula. In fact, the new association and its multiplicity constraints force the students to be the exactly as many as the curricula, which is impossible. Observe that, with a logical formalization the UML class diagram, one can actually perform such form of reasoning making use of automated reasoning tools[1].

[1] Actually, current CASE tools do not perform any kind of automated reasoning on UML class diagrams yet.

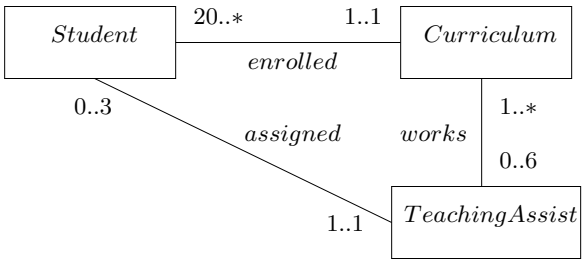

Fig. 3. Another finitely inconsistent class diagram.

Referring to Figure 2, note that the multiplicity constraints do not rule out the possibility of having *infinitely many* students and curricula. When a class is forced to have either zero or infinitely many instances, it is said to be *finitely inconsistent* or *finitely unsatisfiable*. For the sake of completeness, we mention that in some situations involving ISA hierarchies (not shown for brevity), classes may be forced to have zero objects, and are thus said to be inconsistent or unsatisfiable in the *unrestricted* sense.

Unsatisfiability, either finite or unrestricted, of a class is a symptom of a bug in the analysis phase, since such a class is clearly superfluous. In particular, finite unsatisfiability is especially relevant in the context of applications, e.g., databases, in which the number of instances is intrinsically finite.

Obviously the situation described by Figure 2 (in particular, the fact that a curriculum is liked by exactly one student) is not realistic. Anyway, finite inconsistency may arise in more complex situations, cf. e.g. Figure 3. Here, for budget reasons, each curriculum has at most six teaching assistants, and, for enforcing effectiveness of service, each TA is assigned to at most three students. The reason why there must be zero (or infinitely many) students is that eighteen, i.e., three times six, is less than twenty. Global reasoning on the whole class diagram is needed to show finite inconsistency. For large, industrial class diagrams, such reasoning is clearly not doable by hand.

If, in our application we have that the UML class diagram will always be instantiated by a finite number of objects (as often is the case), then, for assessing quality of the analysis phase in software development, we must take fully into account such an assumption. This is reflected in focusing on finite model reasoning.

In this paper we address the implementation of finite model reasoning on UML class diagrams, a task that has not been attempted so far. This is done by exploiting an encoding of UML class diagrams in terms of Description Logics (DLs) [6, 4].

DLs [1] are logics for representing and reasoning on domains of interest in terms of classes and relationships among classes. They are extensively used to formalize conceptual models and object-oriented models in databases and software engineering, and lay the foundations for ontology languages used in the Semantic Web. From a technical point of view, DLs can be seen as multi-modal

logics [16] specifically tailored to capture the typical constructs of class-based formalisms. Alternatively, they can be seen as well-behaved fragments of first-order logic. Indeed, reasoning in such logics has been studied extensively, both from a theoretical point of view, establishing EXPTIME-completeness of various DL variants [9], and from a practical point of view, developing practical reasoning systems. State-of-the-art DL reasoning systems, such as FACT[2] and RACER[3], are highly optimized and result among the best reasoners for modal logics.

The correspondence between UML class diagrams and DLs allows one to use the current state-of-the-art DL reasoning systems to reason on UML class diagrams. However, the kind of reasoning that such systems support is unrestricted (as in first-order logic), and not finite model reasoning. That is, the fact that models (i.e., instantiations of the UML class diagram) must be finite is not taken into account.

Interestingly, in DLs, finite model reasoning has been studied from a theoretical perspective, and its computational complexity has been characterized for various cases [14, 10, 7, 15]. However, no implementations of such techniques have been attempted till now. In this paper we reconsider such work, and on the basis of it we present, to the best of our knowledge, the first implementation of finite model reasoning in UML class diagrams.

The main result of this paper is that it is possible to use off-the-shelf tools for constraint modelling and programming for obtaining a finite model reasoner. In particular, exploiting the finite model reasoning technique for DLs presented in [10, 7], we propose an encoding of UML class diagram satisfiability as a Constraint Satisfaction Problem (CSP). Moreover, we show also how constraint programming can be used to actually return a finite model of the UML class diagram.

We built a system that accepts as input a class diagram written in the MOF syntax, and translates it into a file suitable for ILOG's OPLSTUDIO, which checks satisfiability and returns a finite model, if there is one. The system allowed us to test the technique on the industrial knowledge base CIM, obtaining encouraging results.

The rest of the paper is organized as follows: in Section 2 we briefly describe the main constructs of UML class diagrams in terms of first-order logic. In Sections 3 and 4 we then show how to encode the UML class diagram finite satisfiability problem as a CSP, and, in Section 5 we show how to find, if possible, a finite model of a class diagram, with non-empty classes and associations. Section 6 is devoted to some observations on complexity issues, while our system is described in Section 7. Finally, Section 8 concludes the paper.

2 Formalization of UML Class Diagrams

UML class diagrams allow for modelling, in a declarative way, the static structure of an application domain, in terms of concepts and relations between them.

[2] `http://www.cs.man.ac.uk/~horrocks/FaCT/`
[3] `http://www.sts.tu-harburg.de/~r.f.moeller/racer/`

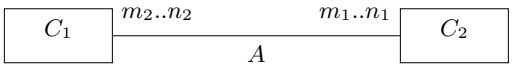

Fig. 4. Binary association in UML with multiplicity constraints.

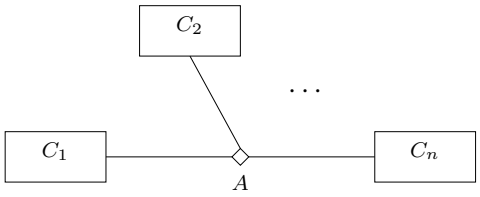

Fig. 5. n-ary association in UML.

We briefly describe UML class diagrams, and specify the semantics of the main constructs in terms of first-order logic (FOL).

A *class* in a UML class diagram denotes a set of objects with common features. Names of classes are unique in a UML class diagram. Formally, a class C corresponds to a FOL unary predicate C. Classes may have attributes and operations, but for simplicity we do not describe them here, since they have only minor impact on the reasoning process.

An *association* in UML is a relation between the instances of two or more classes. Names of associations are unique in a UML class diagram. A binary association A between two classes C_1 and C_2 is graphically rendered as in Figure 4. The *multiplicity* $m_1..n_1$ on the binary association specifies that each instance of the class C_1 can participate at least m_1 times and at most n_1 times to A, similarly for C_2. When the multiplicity is omitted, it is intended to be $0..*$. Observe that an association can also relate several classes $C_1, C_2, \ldots, C_n$, as depicted in Figure 5[4].

An association A between the instances of classes $C_1, \ldots, C_n$, can be formalized as an n-ary predicate A that satisfies the following FOL assertion:

$$\forall x_1, \ldots, x_n.\ A(x_1, \ldots, x_n) \rightarrow C_1(x_1) \wedge \ldots \wedge C_n(x_n)$$

For binary associations (see Figure 4), multiplicities are formalized by the FOL assertions:

$$\forall x.\ C_1(x) \rightarrow (m_1 \leq \sharp\{y \mid A(x,y)\} \leq n_1)$$
$$\forall y.\ C_2(y) \rightarrow (m_2 \leq \sharp\{x \mid A(x,y)\} \leq n_2)$$

where we have abbreviated FOL formulas expressing cardinality restrictions.

Aggregations, which are a particular kind of binary associations are modeled similarly.

[4] In UML, differently from other conceptual modelling formalisms, such as Entity-Relationship diagrams [2], multiplicities are look-across cardinality constraints [17]. This makes their use in non-binary associations difficult with respect to both modelling and reasoning.

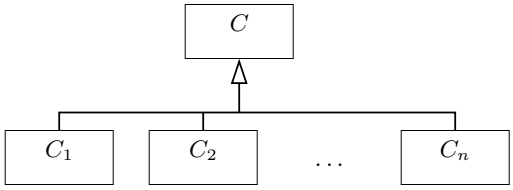

Fig. 6. A class hierarchy in UML.

In UML one can use a *generalization* between a parent class and a child class to specify that each instance of the child class is also an instance of the parent class. Hence, the instances of the child class inherit the properties of the parent class, but typically they satisfy additional properties that in general do not hold for the parent class. Several generalizations can be grouped together to form a *class hierarchy* (also called *ISA hierarchy*), as shown in Figure 6. *Disjointness* and *completeness constraints* can also be enforced on a class hierarchy (graphically, by adding suitable labels). A class hierarchy is said to be disjoint if no instance can belong to more than one derived class, and complete if any instance of the base class belongs also to some of the derived classes.

A UML class C generalizing a class C_1 can be formally captured by means of the FOL assertion:

$$\forall x.\ C_1(x) \rightarrow C(x)$$

A class hierarchy as the one in Figure 6 is formally captured by means of the FOL assertions:

$$\forall x.\ C_i(x) \rightarrow C(x), \qquad \text{for } i = 1, \ldots, n$$

Disjointness among $C_1, \ldots, C_n$ is expressed by the FOL assertions

$$\forall x.\ C_i(x) \rightarrow \textstyle\bigwedge_{j=i+1}^{n} \neg C_j(x), \qquad \text{for } i = 1, \ldots, n-1$$

The *completeness constraint* expressing that each instance of C is an instance of at least one of $C_1, \ldots, C_n$ is expressed by:

$$\forall x.\ C(x) \rightarrow \textstyle\bigvee_{i=1}^{n} C_i(x)$$

In UML class diagrams, it is typically assumed that all classes not in the same hierarchy are a priori disjoint. Similarly, it is typically assumed that objects in a hierarchy must belong to a single most specific class. Hence, two classes in a hierarchy may have common instances only if they have a common subclass.

3 Finite Model Reasoning on UML Class Diagrams

As mentioned before, a technique for finite model reasoning in UML class diagrams can be derived from techniques developed in the context of Description Logics (DLs) [1]. Such techniques are based on translating a DL knowledge base

into a set of linear inequalities [10, 7]. The first-order formalization of UML class diagrams shown in the previous section can be rephrased in terms of DLs. Hence, the finite model reasoning techniques for DLs can be used also for UML class diagrams.

Intuitively, consider a simple UML class diagram D with only binary associations, and in which we do not make use of generalization and hierarchies. Further, for each association, between two classes multiplicities are specified. Figure 4 shows a fragment of such a diagram, in which we have two classes C_1 and C_2 and an association A between them. The multiplicities in the figure express that each instance of C_1 is associated with at least m_1 and at most n_1 instances of C_2 through the association A, similarly for C_2. It is easy to see that such a class diagram D is always satisfiable (assuming $m_i \leq n_i$) if we admit infinite models. Hence, only finite model reasoning is of interest. We observe that, if D is finitely satisfiable, then it admits a finite model in which all classes are pairwise disjoint. Exploiting this property, we can encode finite satisfiability of D in a constraint system as follows. We introduce one variable for each class and one for each association, representing the number of instances of the class (resp., association) in a possible model of D. Then, for each association A we introduce the constraints:

$$\begin{aligned} m_1 * c_1 \leq a \leq n_1 * c_1 \\ m_2 * c_2 \leq a \leq n_2 * c_2 \\ c_1 * c_2 \geq a \end{aligned}$$

where c_1, c_2, and a are the variables corresponding to C_1, C_2, and A, respectively.

It is possible to show that, from a solution of such a constraint system we can construct a finite model of D in which the cardinality of the extension of each class and association is equal to the value assigned to the corresponding variable[5] [14].

The above approach can be extended to deal also with generalizations, disjointness, and covering between classes. Intuitively, one needs to introduce one variable for each combination of classes; similarly, for associations one needs to distinguish how, among the possible combinations of classes, the association is typed in its first and second component. This leads, in general, to the introduction of an exponential number of variables and constraints [10, 7]. We illustrate this in the next section.

4 Finite Model Reasoning via CSP

We address the problem of finite satisfiability of UML class diagrams, and show how it is possible to encode two problems as constraint satisfaction problems (CSPs), namely:

1. deciding whether all classes in the diagram are simultaneously finitely satisfiable, and

[5] In fact, if one is interested just in the existence of a finite model, one could drop the nonlinear constraints of the form $c_1 * c_2 \geq a$.

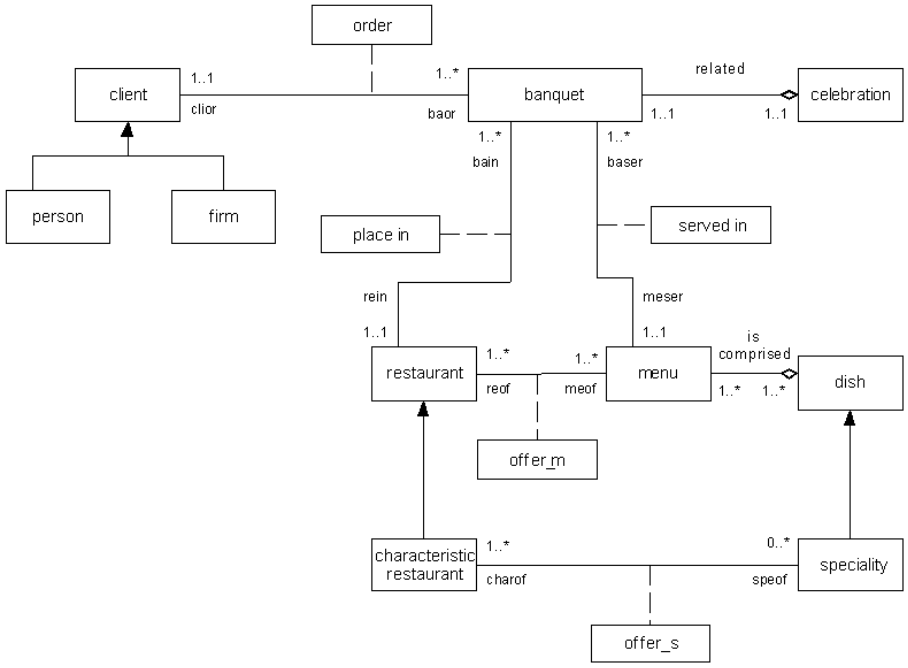

Fig. 7. The "restaurant" UML class diagram.

2. finding –if possible– a finite model with non-empty classes and associations.

We use the "restaurant" class diagram, shown in Figure 7, as our running example.

First we address the problem of deciding finite satisfiability. As mentioned before, we use the technique proposed in [7], which is based on the idea of translating the multiplicity constraints of the UML class diagram into a set of inequalities among integer variables.

The variables and the inequalities of the CSP are modularly described considering in turn each association of the class diagram. Let a be an association between classes *c1* and *c2* such that the following multiplicity constraints are stated:

- There are at least *min_c1* and at most *max_c1* links of type a (instances of the association a) for each object of the class *c1*;
- There are at least *min_c2* and at most *max_c2* links of type a for each object of the class *c2*.

Referring to Figure 7, if a stands for *served_in*, *c1* stands for *banquet*, and *c2* stands for *menu*, then *min_c1* is 1, *max_c1* is 1, *min_c2* is 1, and *max_c2* is ∞.

For the sake of simplicity, we start from the special case in which neither *c1* nor *c2* participates in a ISA hierarchy, e.g., the *related* and the *served_in* associations of Figure 7.

The CSP is defined as follows:

- There are three non-negative variables `c1`, `c2`, and `a`, which stand for the number of objects of the classes and the number of links, respectively (in practice, upper bounds for these variables can be set to a huge constant, e.g., `maxint`);
- There are the following constraints (we use the syntax of the constraint programming language OPL [18]):

```
1. min_c1 * c1 <= a;
2. max_c1 * c1 >= a;
3. min_c2 * c2 <= a;
4. max_c2 * c2 >= a;
5. a  <= c1 * c2;
6. a  >= 1;
7. c1 >= 1;
8. c2 >= 1;
```

Constraints 1–4 account for the multiplicity of the association; they can be omitted if either $\texttt{min\_} = 0$, or $\texttt{max\_} = \infty$ (symbol '`*`' in the class diagram). Constraint 5 sets an upper bound for the number of links of type *a* with respect to the number of objects. Constraints 6–8 define the satisfiability problem we are interested in: we want at least one object for each class and at least one link for each association. The latter constraints can be omitted by declaring the variables as strictly positive. Finally, to avoid the system returning an ineffectively large solution, an objective function that , e.g., minimizes the overall number of objects and links, may be added. However, the addition of such or other objective functions is out of the scope of this paper.

When either *c1* or *c2* are involved in ISA hierarchies, the constraints are more complicated, because the meaning of the multiplicity constraints changes. As an example, the multiplicity `1..*` of the *order* association in Figure 7 states that a *client* order*s* at least one *banquet*, but the client can be a *person*, a *firm*, both, or neither (assuming the generalization is neither disjoint nor complete). In general, for an ISA hierarchy involving n classes, $O(2^n)$ non-negative variables corresponding to all possible combinations must be considered. For the same reason, we must consider four distinct specializations of the *order* association, i.e., one for each possible combination. Summing up, we have the following non-negative variables:

- `person`, `order_p`, for clients who are persons and not firms;
- `firm`, `order_f`, for clients who are firms and not persons;
- `person_firm`, `order_pf`, for clients who are both firms and persons;
- `client`, `order_c`, for clients who are neither firms nor persons;

plus the positive `banquet` variable.

The constraints (in the OPL syntax) which account for the *order* association are as follows:

```
/*  1 */  client <= order_c;
/*  2 */  firm <= order_f;
/*  3 */  person <= order_p;
/*  4 */  person_firm <= order_pf;
/*  5 */  banquet = order_c + order_f + order_p + order_pf;
/*  6 */  order_c <= client * banquet;
/*  7 */  order_f <= firm * banquet;
/*  8 */  order_p <= person * banquet;
/*  9 */  order_pf <= person_firm * banquet;
/* 10 */  client + firm + person + person_firm >= 1;
/* 11 */  order_c + order_f + order_p + order_pf >= 1;
```

Constraints 1–4 account for the '1' in the `1..*` multiplicity; Constraint 5 translates the `1..1` multiplicity; Constraints 6–9 set an upper bound for the number of links of type *order* with respect to the number of objects; Constraints 10–11 define the satisfiability problem (`banquet` is already strictly positive).

We refer the reader to [7, 8] for formal details of the translation, and in particular for the proof of its correctness. As for the implementation, the "restaurant" example has been encoded in OPL as a CSP with 24 variables and 40 constraints. The solution has been found by the underlying constraint programming solver, i.e., ILOG's SOLVER [12, 11], in less than 0.01 seconds.

5 Constructing a Finite Model

We now turn to the second problem, i.e., finding –if possible– a finite model with non-empty classes and associations. The basic idea is to encode in the constraint modelling language of OPL the semantics of the UML class diagram (see Section 2 and [4, 3]). In particular we use arrays of boolean variables representing the extensions of predicates, where the size of the arrays is determined by the output of the first problem. Since in the first problem we have enforced the multiplicity constraints, and obtained an admissible number of objects for each class, we know that a finite model of the class diagram exists, and we also know the size of the universe of such a finite model, which is the sum of the objects of the classes.

Referring to our "restaurant" example, we have the following declarations describing the size of our universe and two sorts:

```
int size = client + person + firm + person_firm + restaurant + menu +
 characteristic_restaurant + dish + specialty + banquet + celebration;
range Bool 0..1;
range Universe 1..size;
```

where `client`, `person` etc. are the number of objects for each class, obtained as the output of the first problem.

The arrays corresponding, e.g., to the *client* and *banquet* classes, and to the *order_c* association are declared as follows:

```
var Bool Client[Universe];
```

```
var Bool Banquet[Universe];
var Bool Order_C[Universe,Universe];
```

Now, we have to enforce some constraints to reflect the semantics of the UML class diagram [4, 3], namely that:

1. Each object belongs to exactly one most specific class;
2. The number of objects (resp., links) in each class (resp., association) is coherent with the solution of the first problem;
3. The associations are *typed*, e.g., that a link of type *order_c* insists on an object which is a *banquet* and on another object which is a *client*;
4. The multiplicity constraints are satisfied.

Such constraints can be encoded as follows (for brevity, we show only some of the constraints).

```
// AN OBJECT BELONGS TO ONE CLASS
forall(x in Universe)
  Client[x] + Person[x] + Firm[x] + Person_Firm[x] + Restaurant[x] +
  Characteristic_Restaurant[x] + Dish[x] + Specialty[x] + Banquet[x] +
  Celebration[x] + Menu[x] = 1;
// ENFORCING SIZES OF CLASSES AND ASSOCIATIONS
sum(x in Universe) Client[x] = client;
sum(x in Universe) Banquet[x] = banquet;
sum(x in Universe, y in Universe) Order_C[x,y] = order_c;
// TYPES FOR ASSOCIATIONS
forall(x, y in Universe)
  Order_C[x,y] => Client[x] & Banquet[y];
// MULTIPLICITY CONSTRAINTS ARE SATISFIED
forall(x in Universe)
  Client[x] => sum(y in Universe) Order_C[x,y] >= 1;
```

Summing up, the "restaurant" example has been encoded in OPL with about 40 lines of code. After instantiation, this resulted in a CSP with 498 variables and 461 constraints. The solution has been found by ILOG's SOLVER in less than 0.01 seconds, and no backtracking.

6 Notes on complexity

Few notes about the computational complexity are in order. It is known that solving both problems of deciding finite satisfiability and finite model finding are EXPTIME-complete [7]. Our encoding of the first problem in a CSP may result in a number of variables which is exponential in the size of the diagram. Anyway, since the exponentiality depends on the maximum number of classes involved in the same ISA hierarchy, the actual size for real UML diagrams will typically not be very large (especially when the most specific class assumption is enforced, cf. Section 2). As for the second problem, our encoding is polynomial in the size of the class diagram. Note that this does not contradict the EXPTIME

lower bound, due to the program complexity of modelling languages such as OPL. Indeed, in [5] it is shown that the program complexity of boolean linear programming is NEXPTIME-hard.

7 Implementation

In this section we describe a system realized in order to automatically produce, given a UML class diagram as input, a constraint-based specification that decides its finite satisfiability (full handling of ISA hierarchies among classes and associations is currently under development).

Two important choices have to be made in order to design such a system: the input language for UML class diagrams, and the output constraint language. As for the former, we decided to use a textual representation of UML class diagrams. To this end, we relied on the standard language "Managed Object Format" (MOF)[6]. To give the intuition of the language, a MOF description of the class diagram depicted in Figure 7 is shown in Figure 8. For what concerns the output language, instead, in order to use state-of-the-art solvers, we opted for the constraint programming language OPL.

However, in order to have a strong decoupling between the two front-ends of the system, we realized it in two, strongly decoupled modules: the first one acts as a server, receiving a MOF file as input and returning a high-level, object-oriented complete internal representation of the described class diagram. A client module, then, traverses the internal model in order to produce the OPL specification.

In this way, we are able to change the language for the input (resp., output) by modifying only the MOF parser (resp., the OPL encoder) module of the system. Moreover, by decoupling the parsing module from the encoder into OPL, we are able to realize new tools to make additional forms of reasoning at a very low cost.

As for the handling of ISA hierarchies, as described in Section 4, an exponential blow-up of the number of variables (one for each combination of classes involved in the hierarchy) cannot be avoided in general. However, in case the hierarchy is disjoint or complete, it is possible to reduce the number of generated variables. It can be observed that, if an ISA is complete, the variable relative to its base class can be avoided; even more interestingly, if an ISA is disjoint, all variables that model instances that belong to *any* combination of two or more derived classes can be ignored, thus reducing the overall number of variables to the number of classes in the hierarchy.

The MOF language provides the "abstract" qualifier that, when applied to a class C imposes that no instance may exist that belongs to class C and does not belong to any of its subclasses. This is implicitly used to assert completeness of the ISA hierarchy. Also, in MOF, the most specific class assumption is implicitly enforced. Hence, in all experiments reflect such an assumption.

In order to test whether using off-the-shelf tools for constraint programming is effective to decide finite satisfiability of real-world UML class diagrams, we

[6] `http://www.dmtf.org`

used our system to produce OPL specifications for several class diagrams of the "Common Information Model" (CIM)[7], a standard model used for describing overall management information in a network/enterprise environment. We don't describe the model into details, since this is out of the scope of the paper. We just observe that the class diagrams we used were composed of about 1000 classes and associations, and so can be considered good benchmarks to test whether current constraint programming solvers can be effectively used to make the kind of reasoning shown so far.

Constraint specifications obtained by giving large class diagrams in the CIM collection, were solved very efficiently by OPL. As an example, when the largest diagram, consisting of 980 classes and associations, has been given as input to our system, we obtained an OPL specification consisting of a comparable number of variables and 862 constraints. Nonetheless, OPL solved it in less than 0.03 seconds of CPU time, by invoking ILOG SOLVER. This high efficiency is achieved also because generated constraints are often easily satisfiable (cardinality constraints for associations often have "0" or "1" as lower bounds, or "$*$" as upper bounds). This is encouraging evidence that current CP technology can be effectively used in order to make finite model reasoning on real-world class diagrams.

8 Conclusions

Finite model reasoning in UML class diagrams, e.g., checking whether a class is forced to have either zero or infinitely many objects, is of crucial importance for assessing quality of the analysis phase in software development. Despite the fact that finite model reasoning is often considered more important than unrestricted reasoning, no implementation of this task has been attempted so far.

In this paper we showed that it is possible to use off-the-shelf tools for constraint modelling and programming for obtaining a finite model reasoner. In particular, exploiting finite model reasoning techniques published previously, we proposed an encoding as a CSP of UML class diagram satisfiability. Moreover, we showed also how constraint programming can be used to actually return a finite model of a class diagram.

We implemented a system which parses class diagrams written in the MOF language and uses ILOG's SOLVER for solving the finite satisfiability problem. The results of the experimentation performed on the CIM class diagram, a large industrial knowledge base, are encouraging, since determining satisfiability is done in just few hundredths of second for a class diagram with 980 classes and associations.

References

1. F. Baader, D. Calvanese, D. McGuinness, D. Nardi, and P. F. Patel-Schneider, editors. *The Description Logic Handbook: Theory, Implementation and Applications.* Cambridge University Press, 2003.

[7] `http://www.dmtf.org/standards/cim`

2. C. Batini, S. Ceri, and S. B. Navathe. *Conceptual Database Design, an Entity-Relationship Approach*. Benjamin and Cummings Publ. Co., Menlo Park, California, 1992.
3. D. Berardi, A. Calì, D. Calvanese, and G. De Giacomo. Reasoning on UML class diagrams. Technical Report 11-03, Dipartimento di Informatica e Sistemistica, Università di Roma "La Sapienza", 2003.
4. D. Berardi, D. Calvanese, and G. De Giacomo. Reasoning on UML class diagrams is EXPTIME-hard. In *Proceedings of the 2003 Description Logic Workshop (DL 2003)*, pages 28–37. CEUR Electronic Workshop Proceedings, `http://ceur-ws.org/Vol-81/`, 2003.
5. M. Cadoli. The expressive power of binary linear programming. In *Proc. of the 7th Int. Conf. on Principles and Practice of Constraint Programming (CP 2001)*, volume 2239 of *Lecture Notes in Computer Science*, 2001.
6. A. Calì, D. Calvanese, G. De Giacomo, and M. Lenzerini. A formal framework for reasoning on UML class diagrams. In *Proceedings of the Thirteenth International Symposium on Methodologies for Intelligent Systems (ISMIS 2002)*, volume 2366 of *Lecture Notes in Computer Science*, pages 503–513. Springer, 2002.
7. D. Calvanese. Finite model reasoning in description logics. In *Proceedings of the Fifth International Conference on the Principles of Knowledge Representation and Reasoning (KR'96)*, pages 292–303, 1996.
8. D. Calvanese. *Unrestricted and Finite Model Reasoning in Class-Based Representation Formalisms*. PhD thesis, Dipartimento di Informatica e Sistemistica, Università di Roma "La Sapienza", 1996. Available at `http://www.dis.uniroma1.it/pub/calvanes/thesis.ps.gz`.
9. D. Calvanese, G. De Giacomo, M. Lenzerini, and D. Nardi. Reasoning in expressive description logics. In A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*, chapter 23, pages 1581–1634. Elsevier Science Publishers (North-Holland), Amsterdam, 2001.
10. D. Calvanese and M. Lenzerini. On the interaction between ISA and cardinality constraints. In *Proceedings of the Tenth IEEE International Conference on Data Engineering (ICDE'94)*, pages 204–213, Houston (Texas, USA), 1994. IEEE Computer Society Press.
11. ILOG Solver system version 5.1 user's manual, 2001.
12. ILOG OPL Studio system version 3.6.1 user's manual, 2002.
13. I. Jacobson, G. Booch, and J. Rumbaugh. *The Unified Modeling Language User Guide*. Addison Wesley Publ. Co., Reading, Massachussetts, 1998.
14. M. Lenzerini and P. Nobili. On the satisfiability of dependency constraints in entity-relationship schemata. *Information Systems*, 15(4):453–461, 1990.
15. C. Lutz, U. Sattler, and L. Tendera. The complexity of finite model reasoning in description logics. In *Proceedings of the Nineteenth International Conference on Automated Deduction (CADE 2003)*, pages 60–74, 2003.
16. K. Schild. A correspondence theory for terminological logics: Preliminary report. In *Proceedings of the Twelfth International Joint Conference on Artificial Intelligence (IJCAI'91)*, pages 466–471, 1991.
17. B. Thalheim. Fundamentals of cardinality constraints. In G. Pernoul and A. M. Tjoa, editors, *Proceedings of the Eleventh International Conference on the Entity-Relationship Approach (ER'92)*, pages 7–23. Springer, 1992.
18. P. Van Hentenryck. *The OPL Optimization Programming Language*. The MIT Press, 1999.

```
class Client { /* Properties and methods (always omitted) */ };
class Person : Client {};
class Firm : Client {};
class Banquet {};
class Celebration {};
class Restaurant {};
class Menu {};
class Dish {};
class CharacteristicRestaurant : Restaurant {};
class Speciality : Dish {};

[association] class Order {
  [Min (1), Max (1)] Client REF clior;
  [Min (1)] Banquet REF baor;
};
[association] class Related {
  [Min (1), Max (1)] Celebration REF cerel;
  [Min (1), Max (1)] Banquet REF barel;
};
[association] class PlaceIn {
  [Min (1)] Banquet REF bain;
  [Min (1), Max (1)] Restaurant REF rein;
};
[association] class ServedIn {
  [Min (1)] Banquet REF baser;
  [Min (1), Max (1)] Menu REF meser;
};
[association] class IsComprised {
  [Min (1)] Menu REF mecom;
  [Min (1)] Dish REF dicom;
};
[association] class OfferM {
  [Min (1)] Restaurant REF reof;
  [Min (1)] Menu REF meof;
};
[association] class OfferS {
  Speciality REF speof;
  [Min (1)] CharacteristicRestaurant REF charof;
};
```

Fig. 8. MOF description of the class diagram in Figure 7.