Finite Model Reasoning on UML Class Diagrams via Constraint Programming*†

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

[1] Dip. di Informatica e Sistemistica
Università di Roma "La Sapienza"
Via Ariosto 25, 00185 Roma, Italy
`cadoli|degiacomo|tmancini@dis.uniroma1.it`

[2] Faculty of Computer Science
Free University of Bozen-Bolzano
Piazza Domenicani 3, 39100 Bolzano, Italy
`calvanese@inf.unibz.it`

Abstract

Finite model reasoning in UML class diagrams is an important task for assessing the quality of the analysis phase in the development of software applications in which it is assumed that the number of objects of the domain is finite. In this paper, we show how to encode finite model reasoning in UML class diagrams as a constraint satisfaction problem (CSP), exploiting techniques developed in description logics. In doing so we set up and solve an intermediate CSP problem to deal with the explosion of "class combinations" arising in the encoding. To solve the resulting CSP problems we rely on the use of

*This research has been partially supported by FET project TONES (Thinking ONtologiES), funded within the EU 6th Framework Programme under contract FP6-7603, and by the PRIN 2006 project NGS (New Generation Search), funded by MIUR.

†R. Basili and M.T. Pazienza (Eds.): AI*IA 2007, LNAI 4733, pp. 36–47, 2007. © Springer-Verlag Berlin Heidelberg 2007

off-the-shelf tools for constraint modeling and programming. As a result, we obtain, to the best of our knowledge, the first implemented system that performs finite model reasoning on UML class diagrams.

1 Introduction

The Unified Modelling Language (UML, [8], 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 among classes; *ISA hierarchies* among classes, i.e., relations establishing that each object of a class is also an object of another class; and *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(a), 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(a), 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

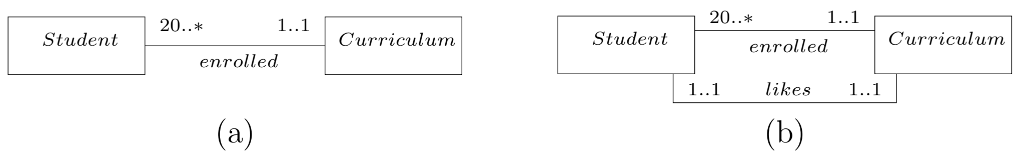

Figure 1: UML class diagrams with (a) finitely satisfiable and (b) finitely unsatisfiable classes.

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) a further association, *likes*, with the constraints that each student likes exactly one curriculum, and that each curriculum is liked by exactly one student (cf., Figure 1(b)), 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 exactly as many as the curricula, which is impossible. Observe that, with a logical formalization of the UML class diagram, one can actually perform such a form of reasoning making use of automated reasoning tools[1].

Referring to Figure 1(b), 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 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 unsatisfiable in the *unrestricted* sense. The above example shows that UML class diagrams do *not have the finite model property*, since unrestricted and finite satisfiability are different.

Unsatisfiability, either finite or unrestricted, of a class is a symptom of a bug in the analysis phase, since either such a class is superfluous, or a conflict has arisen while modeling different, antithetic, requirements. In particular, finite unsatisfiability is especially relevant in the context of applications, e.g., databases, in which the number of instances is intrinsically finite. Global reasoning on the whole class diagram is needed to show finite unsatisfiability. For large, industrial class diagrams, finite unsatisfiability could easily arise, because different parts of the same diagram may be synthesized by different

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

analysts, and is likely to be nearly impossible to be discovered by hand.

In this paper, we address finite model reasoning on UML class diagrams, a task that, to the best of our knowledge, has not been attempted so far. This is done by exploiting an encoding of UML class diagrams in terms of Description Logics (DLs) [2], in order to take advantage of the finite model reasoning techniques developed for DLs [4, 5].These techniques, which are optimal from the computational complexity point of view, are based on a reduction of reasoning on a DL knowledge base to satisfaction of linear constraints.

The contribution of this paper is on the practical realization of such finite modeling reasoning techniques by making use of off-the-shelf tools for constraint modelling and programming. In particular, by exploiting the finite model reasoning technique for DLs presented in [4, 5], we propose an encoding of UML class diagram satisfiability as a Constraint Satisfaction Problem (CSP). We show that, in spite of the high computational complexity of the reasoning task in general, the aforementioned techniques are feasible in practice, if some optimizations are performed in order to reduce the exponential number of variables in the constraint problem. We do so by relying again on the constraint solver itself, by setting up and solving an auxiliary constraint problem that exploits the structure of real-world UML class diagrams.

We built a system that accepts as input an UML class diagram (written in the standard MOF syntax[2]), and reasons on it according to the ideas above making use of the ILOG's OPLSTUDIO constraint system. The system allowed us to test the technique on the industrial knowledge base CIM.

2 Description Logics

DLs [1] are logics for representing a domain of interest in terms of classes and relationships among classes and reasoning on it. They are extensively used to formalize conceptual models and object-oriented models in databases and software engineering [3, 2], and lay the foundations for ontology languages used in the Semantic Web.

In DLs, the domain of interest is modeled through *concepts*, denoting classes of objects, and *roles*, denoting binary relations between objects. The semantics of DLs is given in terms of an *interpretation* $\mathcal{I} = (\Delta^{\mathcal{I}}, \cdot^{\mathcal{I}})$ consisting of an interpretation *domain* $\Delta^{\mathcal{I}}$ and an *interpretation function* $\cdot^{\mathcal{I}}$ that maps

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

Syntax	**Semantics**
$\neg B$	$\Delta^{\mathcal{I}} \setminus B^{\mathcal{I}}$
$D_1 \sqcap D_2$	$D_1^{\mathcal{I}} \cap D_2^{\mathcal{I}}$
$D_1 \sqcup D_2$	$D_1^{\mathcal{I}} \cup D_2^{\mathcal{I}}$
$\forall R.D$	$\{a : \forall b.\, (a,b) \in R^{\mathcal{I}} \rightarrow b \in D^{\mathcal{I}}\}$
$(\geq m\, R)$	$\{a : \lvert\{b : (a,b) \in R^{\mathcal{I}}\}\rvert \geq m\}$
$(\leq n\, R)$	$\{a : \lvert\{b : (a,b) \in R^{\mathcal{I}}\}\rvert \leq n\}$
P^-	$\{(a,b) : (b,a) \in P^{\mathcal{I}}\}$

Figure 2: Syntax and semantics of $\mathcal{ALUNI}$

every concept D to a subset $D^{\mathcal{I}}$ of $\Delta^{\mathcal{I}}$ and every role R to a subset $R^{\mathcal{I}}$ of $\Delta^{\mathcal{I}} \times \Delta^{\mathcal{I}}$. In this paper we deal with the DL $\mathcal{ALUNI}$ [4, 5], whose syntax and semantics are shown in Figure 2 (B and P denote respectively atomic concepts and roles, D and R respectively arbitrary concepts and roles, m a positive integer, and n a non-negative integer). The constructs $(\geq m\, R)$ and $(\leq n\, R)$ are called *number restrictions*. We refer to [1] for more details on DLs.

An $\mathcal{ALUNI}$ knowledge base (KB) is constituted by a finite set of *(primitive) inclusion assertions* of the form $B \sqsubseteq D$. An interpretation $\mathcal{I}$ is called a *model* of a KB if $B^{\mathcal{I}} \subseteq D^{\mathcal{I}}$ for each assertion $B \sqsubseteq D$ in the KB. The basic reasoning tasks in DLs are (finite) KB and concept satisfiability: a KB is *(finitely) satisfiable* if it admits a (finite) model; a concept C is *(finitely) satisfiable* in a KB, if the KB admits a (finite) model $\mathcal{I}$ such that $C^{\mathcal{I}} \neq \emptyset$.

Due to the expressiveness of the constructs present in $\mathcal{ALUNI}$ KBs, unrestricted and finite satisfiability are different problems, i.e., $\mathcal{ALUNI}$ does not have the *finite model property* (cf. [5]). Unrestricted model reasoning is a quite well investigated problem in DLs, and several DL reasoning systems that perform such kind of reasoning are available (e..g, Fact++[3] or Racer[4]).

Instead, finite model reasoning is less well studied, both from the theoretical and from the practical point of view. To the best of our knowledge, no implementation of finite model reasoners has been attempted till now. Some works provide theoretical results showing that finite model reasoning over a KB can be done in EXPTIME for variants of expressive DLs, including $\mathcal{ALUNI}$ [4, 5, 10]. Notice that this bound is tight, since (finite) model

[3]`http://owl.man.ac.uk/factplusplus/`

[4]`http://www.racer-systems.com/`

reasoning is already EXPTIME-hard even for much less expressive DLs (enjoying the finite model property) [1]. These results are based on an encoding of the finite model reasoning problem into the problem of finding particular integer solutions to a system of linear inequalities. Such solutions can be put in a direct correspondence with models of the KB in which the values provided by the solution correspond to the cardinalities of the extensions of concepts and roles. Also, the specific form of the system of inequalities guarantees that the existence of an arbitrary solution implies the existence of an integer solution. Moreover, from the encoding it is possible to deduce the existence of a bound on the size of an integer solution, as specified by the following theorem.

Theorem 1 ([5]) *Let $\mathcal{K}$ be an $\mathcal{ALUNI}$ KB of size K, C an atomic concept, $\Psi_{\mathcal{K},C}$ the system of linear inequalities derived from $\mathcal{K}$ and C, and N the maximum number appearing in number restrictions in $\mathcal{K}$. Then, C is satisfiable in $\mathcal{K}$ if and only if $\Psi\mathcal{K},\mathcal{B}$ admits a solution. Moreover, if a solution exists, then there is one whose values are bounded by $(K \cdot N)^{O(K)}$.*

In the following, we will exploit the above result to derive a technique for reasoning on UML class diagrams that properly takes into account finiteness of the domain of interest. The technique is a based on an encoding of UML class diagrams in terms of DL KBs, which we present in the next section.

3 Formalizing UML Class Diagrams in DLs

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. Here, we briefly describe the core part of UML class diagrams, and specify the semantics of its constructs in terms of $\mathcal{ALUNI}$. An in-depth treatment on the correspondence between UML class diagrams and DLs can be found in [2].

A *class* in a UML class diagram denotes a set of objects with common features. Formally, a class C corresponds to a concept C. Classes may have attributes and operations, but for simplicity we do not consider them here, since they don't play any role in the finite class unsatisfiability problem.

A (binary) *association* in UML is a relation between the instances of two classes. An association A between two classes C_1 and C_2 is graphically rendered as in Figure 3(a). The *multiplicity* $m_1..n_1$ on the binary association

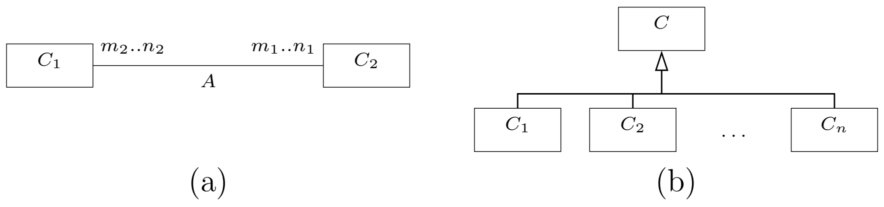

Figure 3: (a) UML binary association with multiplicity constraints. (b) ISA hierarchy.

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. $*$ is used to specify no upper bound. [5]

An association A between the instances of classes C_1 and C_2, can be formalized as an atomic role A characterized by $C_1 \sqsubseteq \forall A.C_2$ and $C_2 \sqsubseteq \forall A^-.C_1$.

For an association as depicted in Figure 3(a), multiplicities are formalized by $C_1 \sqsubseteq (\geq m_1\, A) \sqcap (\leq n_1\, A)$ and $C_2 \sqsubseteq (\geq m_2\, A^-) \sqcap (\leq n_2\, A^-)$.

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 3(b). *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 class C generalizing a class C_1 can be formalized as: $C_1 \sqsubseteq C$. A class hierarchy as shown in Figure 3(b) is captured by $C_i \sqsubseteq C$, for $i = 1, \dots, n$.

Disjointness among $C_1, \dots, C_n$ is expressed by $C_i \sqsubseteq \bigwedge_{j=i+1}^{n} \neg C_j$, for $i = 1, \dots, n-1$. The *completeness constraint* expressing that each instance of C is an instance of at least one of $C_1, \dots, C_n$ is expressed by $C \sqsubseteq \bigsqcup_{i=1}^{n} C_i$.

Here, we follow a typical assumption in UML class diagrams, namely that

[5]In UML, an association can have arbitrary arity and relate several classes, but for simplicity we do not consider this case here (but see Conclusions). *Aggregations*, which are a particular kind of binary associations are modeled similarly to associations.

all classes not in the same hierarchy are a priori disjoint. Another typical assumption, called *unique most specific class assumption*, is that objects in a hierarchy must belong to a single most specific class. Hence, under such an assumption, two classes in a hierarchy may have common instances only if they have a common subclass. We discuss in the next section the effect of making the unique most specific class assumption when reasoning on an UML class diagram.

The basic form of reasoning on UML class diagrams is (finite) satisfiability of a class C, which amounts to checking whether the class diagram admits a (finite) instantiation in which C has a nonempty extension. Formally, this corresponds to checking whether the concept corresponding to C is (finitely) satisfiable in the KB formalizing the diagram. As mentioned, unrestricted and finite satisfiability in UML class diagrams (and also in $\mathcal{ALUNI}$) are different problems.

The formalization of UML class diagrams in terms of DLs [2], and the fact that instantiations of the UML class diagram must be finite, allows one to use on such diagrams the techniques for finite model reasoning in DLs discussed in Section 2. Specifically, the EXPTIME upper bounds apply also to finite model reasoning on UML class diagrams [2]. Instead, the exact lower bound of reasoning on UML class diagrams as presented above is still open. However, if one adds subsetting relations between associations or the ability of specializing the typing of an association for classes in a generalization, then both unrestricted and finite model reasoning are EXPTIME-hard (see [2]). This justifies the approach taken in the next section, where we address the problem of finite model reasoning on UML class diagrams also from a practical point of view.

4 Finite Model Reasoning on UML Class Diagrams via CSP

We address now finite class satisfiability in UML class diagrams, and show how it is possible to encode the problem as a constraint satisfaction problem (CSP).

As mentioned, a technique for finite model reasoning in UML class diagrams can be derived from techniques developed in the context of DLs. Such techniques are based on translating a DL knowledge base into a set of linear inequalities [4, 5]. The formalization of UML class diagrams in terms of DLs implies that the finite model reasoning techniques for the latter can be used also for the former.

In the rest of this paper, we will deal directly with the UML class diagram constructs, considered, from a formal point of view, as abbreviations for the corresponding DL concepts and roles.

Intuitively, consider a simple UML class diagram D with no generalizations and hierarchies. Figure 3(a) shows a fragment of such a diagram, in which we have two classes C_1 and C_2 and an association A between them. 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 class C_1 in D in a constraint satisfaction problem. The variables and the constraints of the CSP are modularly described considering in turn each association of the class diagram. Let A be an association between classes C_1 and C_2 such that the following multiplicity constraints are stated (cf. Figure 3(a)):

- There are at least m_1 and at most n_1 links of type A (instances of the association A) for each object of the class C_1;

- There are at least m_2 and at most n_2 links of type A for each object of C_2.

In the special case in which neither C_1 nor C_2 participates in an ISA hierarchy, 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[6], respectively (upper bounds for these variables follow from Theorem 1; in practice, they can be set to a huge constant, e.g., `maxint`);

- There are the following constraints (we use, here and in what follows, a syntax similar to that of OPL[11]):

[6]The use of variables standing for the number of links stems from the technique proposed in [5], which ensures soundness and completeness of reasoning. It remains to be investigated whether a simpler encoding avoiding the use of such variables is possible.

```
1. m1 * c1 <= a;                3. m2 * c2 <= a;
2. n1 * c1 >= a;                4. n2 * c2 >= a;

                5. a  <= c1 * c2;
                6. c1 >= 1;
```

Constraints 1–4 account for the multiplicity of the association; they can be omitted if either m_1 or m_2 is 0, or n_1 or n_2 is ∞ (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. Constraint 6 encodes satisfiability of class C_1: we want at least one object in its extension. As an example, consider the Restaurant class diagram, shown in Figure 4: if A stands for *served_in*, C_1 stands for *menu*, and C_2 stands for *banquet*, then m_1 is 1, n_1 is ∞, m_2 is 1, and n_2 is 1.

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.

It is possible to show that, from a solution of such a constraint system we can construct a finite model of the class diagram in which the cardinality of the extension of each class and association is equal to the value assigned to the corresponding variable[7] [9].

When either C_1 or C_2 are involved in ISA hierarchies, the constraints are actually more complicated, because the meaning of the multiplicity constraints changes. As an example, the multiplicity `1..*` of the *order* association in Figure 4 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, 2^n non-negative variables corresponding to all possible combinations must be considered. For the same reason, in our example, 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;

[7]In fact, if one is interested just in the existence of a finite model, the nonlinear constraints $a \leq c_1 * c_2$ can be dropped. Indeed, any solution of the resulting constraint system can be transformed into one that satisfies also the nonlinear constraint by multiplying it with a sufficiently large constant, cf. [5].

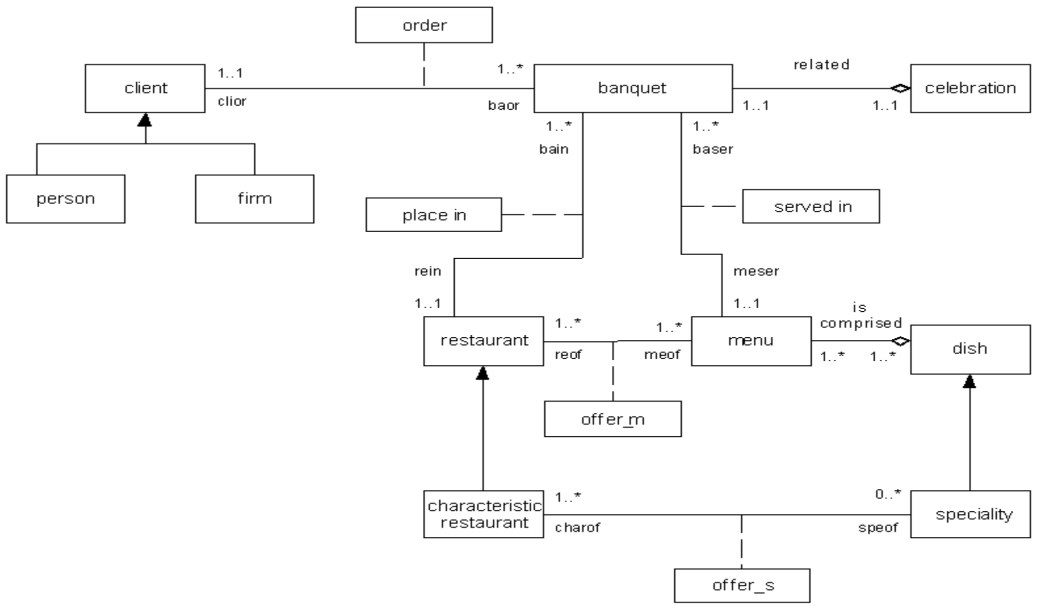

Figure 4: The Restaurant UML class diagram.

- `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 non-negative `banquet` variable.

The constraints 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;
```

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; Constraint 10 encodes satisfiability of the *client* class.

We refer the reader to [5] for formal details of the translation and 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, [7], in less than 0.01 seconds.

The exponential blow-up in the number of variables and constraints due to the presence of ISA hierarchies is a major obstacle when dealing with large class diagrams, such as those describing real-world applications. To this end, special care to reduce the size of the resulting CSP as much as possible is mandatory.

In particular, if a given ISA hierarchy (with C as parent class and $\{C_1, \ldots, C_n\}$ as children) is *complete*, the variable for C can be removed. Moreover, if the ISA is *disjoint*, we can omit all the variables that model instances that belong to *any* combination of two or more derived classes, hence reducing the overall number of variables to the number of classes in the hierarchy. As an example, if the ISA among *Client*, *Person*, and *Firm* in the Restaurant example is complete, variables `client` and `order_c` are superfluous. Similarly, if the ISA is disjoint, variables `person_firm` and `order_pf` can be omitted.

In order to derive the set of combinations of classes (called, in what follows, "types") that may have common instances, we show now that we can use CP technology again. Indeed, for a given UML class diagram, we can set up and solve an auxiliary constraint problem. The constraint problem is defined in such a way that the set of its solutions corresponds to the set of all those types that are consistent with the ISA hierarchies of the diagram, i.e., those types that can be populated without violating any of the constraints expressed by the ISA hierarchies. More precisely, assuming the classes of the diagram are represented by integers between 1 and `nclasses`, the constraint problem is defined as follows (we use again a pseudocode resembling the OPL syntax):

```
Given the set of ISA hierarchies of an UML class diagram
Find boolean legalType[1..nclasses] such that:

For each ISA (C1...Cn is-a C) {
  for each i = 1..n:  legalType[Ci] -> legalType[C];
  If ISA is disjoint:  at_most_one(i = 1..n)(legalType[Ci]);
  If ISA is complete:  legalType[C] -> exists i=[1..n] s.t. legalType[Ci];
}
legalType is a combination of at least one class;
Classes that belong to legalType must be connected by ISA hierarchies;
```

By computing all solutions of this auxiliary constraint problem, we obtain the set of all types that are consistent with the ISA hierarchies. Given a solution `legalType[]` (an array of booleans), the corresponding type is made of all classes `C` such that `legalType[C] = true`). Only variables for types found in this way need to be generated in order to solve the finite satisfiability problem. It is worth noting that in practical circumstances, the number of all possible types is not expected to be huge. In fact, well designed class diagrams, even if the unique most specific class assumption is not made (cf. end of Section 3), usually have a small amount of non-disjoint ISAs, since this helps to increase the overall quality of the diagram, by making the partitions of concepts that are important for the application explicit. Some experimental results that show the applicability of the approach when reasoning on real-world class diagrams are described in Section 5.

Once a UML class diagram is shown to be finitely satisfiable, a second problem is to return a model with non-empty classes and associations. To solve this problem, we can use again constraint technology, by writing a constraint program that encodes the semantics of the UML class diagram (cf. Section 3), and uses the output of the finite satisfiability problem to fix the size of the model. In fact, since in the finite satisfiability problem we have enforced the multiplicity constraints, we know that a finite model of the class diagram exists, and we also know an admissible number of instances for each class and association. We do not describe the relevant constraint program for space reasons, but just observe that, for the Restaurant example (encoded in OPL with about 40 lines of code, which 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.

5 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 finite class satisfiability. Two important choices were made in the design phase: the input language for class diagrams, and the output constraint language. As for the former, we decided to use a standard textual representation of UML class diagrams called "Managed Object Format" (MOF) (cf. footnote 2). Concerning 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 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 (actually, the system supports the concepts in the core UML, i.e., classes, associations, hierarchies among classes, and subset relationships between associations). A client module, then, traverses the internal model in order to produce the OPL specification encoding the finite satisfiability problem for the diagram (actually, subset relationships between associations are not taken into account). With this decoupling, 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 low cost.

As for the handling of ISA hierarchies, it has already been mentioned that an exponential blow-up of the number of variables (one for each combination of classes involved in the hierarchy) cannot be avoided in the worst case. However, in case of disjoint or complete hierarchies, it is possible to strongly reduce the number of generated variables (cf. Section 4).

Hence, the system works in two stages. In the first one, after having built the internal representation of the input class diagram, it solves the OPL auxiliary constraint problem described in Section 4 in order to detect all possible combinations of classes (the so-called "types") belonging to the same hierarchy that may have objects in common. In the second stage, it uses this knowledge to build the OPL program that models the finite satisfiability problem for the class diagram.

In order to test whether using off-the-shelf tools for constraint programming is effective to decide finite satisfiability of real-world diagrams, we used our system to produce OPL specifications for several class diagrams of the "Common Information Model" (CIM)[8], a standard model used for describing overall management information in a network/enterprise environment. We don't describe the model in detail, but 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 perform the kind of reasoning shown so far.

Constraint specifications obtained from 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, was 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 of the "structural" aspects usually present in UML class diagrams that model real-world applications. In particular, multiplicity constraints on many associations had "0" or "1" as lower bounds, or "$*$" as upper bounds, and hence the corresponding OPL constraints were easily satisfiable. The consequence is that only a small portion of the constraints of the overall constraint model needed a deep search for finding a solution. Moreover, the exponential explosion of the number of variables for classes belonging to ISA hierarchies was not a problem, since the unique most specific class assumption is implicitly made in these diagrams (hence, non-disjointness among classes was always explicitly stated). This is encouraging evidence that current CP technology can be effectively used in order to make finite model reasoning on real-world class diagrams.

6 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 important for assessing quality of the analysis phase in software development. Despite the importance of finite model reasoning, no implementation of this task has been attempted so far. In this paper we have shown how one can develop such a system by relying on off-the-shelf tools for constraint modeling

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

and programming, using techniques for finite model reasoning in description logics, and putting special care in taming the class-combination explosion.

For simplicity, in this paper we have dealt with binary associations only, but in fact the technique can be straightforwardly extended to n-ary associations[9] as well, and in fact, our current implementation deals also with them.

This paper can also be seen as the first attempt to obtain a practical, computationally optimal finite model reasoner for expressive description logics. Indeed, the techniques developed here apply to $\mathcal{ALUNI}$ knowledge bases with primitive inclusion assertions [6]. More generally, the ideas of applying CSP tools and taking special care in limiting the "class combinations" explosion, could be applied to more expressive description logics as well [10].

References

[1] Franz Baader, Diego Calvanese, Deborah McGuinness, Daniele Nardi, and Peter F. Patel-Schneider, editors. *The Description Logic Handbook: Theory, Implementation and Applications.* Cambridge University Press, 2003.

[2] Daniela Berardi, Diego Calvanese, and Giuseppe De Giacomo. Reasoning on UML class diagrams. *Artificial Intelligence*, 168(1–2):70–118, 2005.

[3] Alexander Borgida, Maurizio Lenzerini, and Riccardo Rosati. Description logics for data bases. In Baader et al. [1], chapter 16, pages 462–484.

[4] Diego Calvanese. Finite model reasoning in description logics. In *Proc. of KR'96*, pages 292–303, 1996.

[5] Diego Calvanese. *Unrestricted and Finite Model Reasoning in Class-Based Representation Formalisms.* PhD thesis, Dip. di Inf. e Sist., Univ. di Roma "La Sapienza", 1996.

[6] Diego Calvanese, Maurizio Lenzerini, and Daniele Nardi. Unifying class-based representation formalisms. *J. of Artificial Intelligence Research*, 11:199–240, 1999.

[9] We do not consider multiplicities for n-ary associations. For a discussion, see [2].

[7] ILOG OPL Studio system version 3.6.1 user's manual, 2002.

[8] Ivar Jacobson, Grady Booch, and James Rumbaugh. *The Unified Modeling Language User Guide.* Addison Wesley Publ. Co., 1998.

[9] Maurizio Lenzerini and Paolo Nobili. On the satisfiability of dependency constraints in entity-relationship schemata. *Information Systems*, 15(4):453–461, 1990.

[10] Carsten Lutz, Ulrike Sattler, and Lidia Tendera. The complexity of finite model reasoning in description logics. In *Proc. of CADE 2003*, pages 60–74, 2003.

[11] Pascal Van Hentenryck. *The OPL Optimization Programming Language.* The MIT Press, 1999.