An Overall Framework for Reasoning About UML/OCL Models Based on Constraint Logic Programming and MDA

Beatriz P´erez
Department of Mathematics and Computer Science,
University of La Rioja,
Logro˜no, Spain.
Email: beatriz.perez@unirioja.es

Ivan Porres
Department of Information Technologies,
˚Abo Akademi University,
Turku, Finland
Email: ivan.porres@abo.fi

Abstract—Due to the widespread adoption of the Model Driven Engineering paradigm, models have become cornerstone components in the software development process. This fact requires verifying such models’ correctness in order to ensure the quality of the final product. In this context, the Unified Modeling Language (UML) and the Object Constraint Language (OCL) constitute two of the most commonly used modeling languages. We have defined an overall framework to reason about UML/OCL models based on Constraint Logic Programming (CLP). In particular, as model finding and design space exploration tool, we use Formula. We show how to translate a UML model into a CLP program following a Meta–Object Facility (MOF) like framework. Furthermore, we enhance our proposal by identifying an expressive fragment of OCL, which guarantees finite satisfiability and we show its translation to Formula. We also complete our approach by developing the CD2Formula Eclipse plug–in, which implements, following a Model Driven Architecture (MDA) approach, our UML model to Formula translation proposal. The proposed framework can be used to reason, validate and verify UML software designs by checking correctness properties and generating model instances using the model exploration tool Formula.

Keywords—UML, OCL, Constraint Logic Programming, reasoning, model verification, MDA

I. INTRODUCTION

This paper is an extension of the work presented in [1]. Due to the widespread adoption of the Model Driven Engineering (MDE) [2] paradigm, models have become cornerstone components in the software development process. This fact requires verifying not only the completeness of such models but also their correctness in order to ensure the quality of the final product, reducing time to market and decreasing development costs. In this context, the Unified Modeling Language (UML) and the Object Constraint Language (OCL) constitute two of the most commonly used modeling languages. On the one hand, UML [3] has been widely accepted as the de–facto standard for building object-oriented software. OCL [4], on the other hand, has been introduced into UML as a logic–based sublanguage to express integrity constraints that UML diagrams cannot convey by themselves.

Unfortunately, in some occasions, possible design errors are not detected until the later implementation stages, thus increasing the cost of the development process [5], [6]. This situation requires a wide adoption of formal methods within the software engineering community. In this line, there have been remarkable efforts to formalize UML semantics to solve ambiguity and under specification detected in UML’s semantics. The formalization and analysis of the specific UML modeled artifacts can be done by carrying out a translation to another language that preserves its semantics [5], [6], [7], [8]. The resulted translation can be used to reason about the software design by checking predefined correctness properties about the original model [6].

In this paper, we propose to use the Constraint Logic Programming (CLP) paradigm as a complementary method for UML modeling foundations, including models’ satisfiability and inspection. More specifically, we focus on UML class diagrams (CD), annotated with OCL constraints, which are considered to be the mainstay of Object-Oriented analysis and design for representing the static structure of a system. Considering CD/OCL models as model representation, we propose an overall framework to reason about such models based on CLP. In particular, as model finding and design space exploration tool we use Formula [9], which stands on algebraic data types (ADT) and CLP, and which has been proved to provide several advantages, including more expressivity, over using other tools [10], [11]. The defined framework is two–fold. Firstly, we have conceptually defined a proposal for the translation of CD/OCL models to Formula. Secondly, we have used a Model Driven Development (MDA) based approach [12] to automatically generate the Formula specification from a class diagram. As for the first contribution, we give a proposal for the translation of a UML model into a Constraint Satisfaction Problem following a multilevel Meta–Object Facility (MOF) like framework. We enhance our proposal by identifying a fragment of OCL that guarantees finite satisfiability, while being, at the same time, considerably expressive. We also show how to translate such OCL fragment to Formula, by giving, as an intermediate step, a representation of the OCL constraints as First-Order Logic (FOL) expressions. As for the second contribution, we have implemented our class diagram to Formula translation approach by using a model-to-text (M2T) transformation tool, obtaining a set of transformation files defined in such a tool. Additionaliy, we have integrated the resulted files into an Eclipse plug–in, called CD2Formula plug–in, we have developed to easily...
and automatically transform a class diagram to Formula. The proposed framework can be used to reason, validate and verify UML software designs by checking correctness properties and generating model instances using the model exploration tool Formula.

As advanced previously, the results presented in this paper are based on the work published by the authors of this paper in [1]. In this paper we provide an extended version of that work, presenting the development of our CD2Formula plug-in as additional contribution.

The remainder of the paper proceeds as follows. In Section II we provide a brief introduction to Formula. An overview of our framework is presented in Section III. Section IV presents the translation of a class diagram to Formula, while Section V describes the chosen OCL fragment and its representation into Formula. The automatic MDA-based translation of a UML class diagram to Formula, together with the development of our CD2Formula plug-in, is presented in Section VI. Section VII summarizes the strengths and weaknesses of our approach and discusses related work. Finally, Section VIII presents our main conclusions and future work.

II. A BRIEF OVERVIEW OF FORMULA

In this section, we provide a general background of the Formula language by presenting the basic Formula concepts. In order to illustrate these basic concepts, we will lean on Figure 1, which, as we will explain later in detail, corresponds to an excerpt of a specific Formula domain we propose to define for translating class diagrams to Formula, but that we use here for explanatory purposes.

A. Formula units and design-space exploration

Formula distinguishes three units for modeling the problem: domains, models and partial models. Modeling in Formula always starts with specifying the problem domain and formalizing an abstraction of the problem that can be used by Formula to reason about the design [13]. A Formula domain FD is the basic specification unit in Formula for an abstraction and allows specifying ADTs and a logic program describing properties of the abstraction. The logic programming paradigm provides a formal and declarative approach for specifying such abstractions [9], which in Formula are represented by rules and queries. A Formula model FM is a finite set of data type instances built from constructors of the associated domain FD, and which satisfies all its constraints [9]. Formula allows to specify individual concrete instances of the design-space or parts thereof, in a specific Formula unit called partial model [9]. A Formula partial model FPM is a set of instance-specific facts placed along with some explicitly mentioned unknowns, which correspond to the parts of the model FM that must be solved. FPMs allow unknowns to be combined with parts of the model that are already fixed [9].

Finally, in order to explore the design-space, Formula loads the specification of the domains and the partial models defined for the specific problem and executes the logic program. The execution finds all intermediate facts that can be derived from the given facts in the partial model, and tries to find valid assignments for the unknowns. This step is carried out by the Formula solver, which, in case it finds a solution that satisfies all encoded constraints, will reconstruct a complete instance model from this information made of known facts [10], [11].

1 domain MetaLevel extends User DataTypes {
  2 Star := {star}.
  3 primitive Class ::= (name: String, isAbstract: Boolean).
  4 primitive Association ::= (name: String, srcType: Class, srcLower:Natural, srcUpper:UpperBound, dstType:Class, dstLower:Natural, dstUpper:UpperBound).
  5 Classifier ::= Class + Association.
  6 errorBadMultInterv := Association(____, srlower, srcUpper, ___).
  7 errorMetaDupAssoc := a1 is Association(name1, ______, ______).
  8 ... 9 conforms := !errorBadMultInterv & !errorMetaDupAssoc & ...}.

Figure 1: An extract of a Formula domain.

B. Domains’ syntax

Basically, a Formula domain consists of abstract data types, rules and queries. Firstly, abstract data types constitute the key syntactic elements of Formula. Based on the defined data types, a number of rules and queries are specified as logic program expressions, ensuring the remaining constraints [9]. Roughly speaking, rules specify implications and queries restrict the valid states by specifying forbidden states.

Abstract data types. They are defined by using the operator ::=, indicating in the right hand side their properties by means of fields. A data type definition can be labeled with the primitive keyword, denoting that it can be used for the extension of other type definitions. Otherwise, the data type results in a derived constructor. As a way of example, in line 3 of Figure 1 we define the Class data type representing the UML Class meta-element constructor. The derived type Classifier, on the other hand, is defined as the union of the Class and Association types (see line 5 of Figure 1).

Around data types, Formula defines different categorizations of the structural elements as building blocks for defining Formula expressions. These elements are mainly Formula terms and predicates.

As it can be inferred from the Help Formula Documentation [13], Formula distinguishes different types of terms, which could be established to be classified into two generalization groups: (1) simple and composite terms, and (2) what they call simply Terms (see Figure 2). On the one hand, Formula defines simple terms and compound terms. A simple term is represented by means of a type identifier containing variables, constants, or other simple terms as arguments, within parenthesis. As a way of example, in line 7 of Figure 1 we show the simple term Association(name1,____,____,____,____), which represents all instances of the Association term, where the first parameter is set to the name property. The other fields of this type (e.g., the srcType, srcLower, srcUpper, dstType, dstLower and dstUpper fields) are filled with a do not-care symbol (‘_’), so that Formula will find valid assignments. A compound term, on the other hand, is represented by means of a type identifier with a list of Terms within parenthesis. As for the other generalization group, on the other hand, the building blocks of Terms are atoms (for
example the identifiers of variables and queries, explained later), arithmetic or aggregation expressions among other terms (sum, count, max, min, etc.), or compound terms.

All these different types of terms are, directly or indirectly, the basis for constructing predicates, which constitute the basic units of data, used for defining queries and rules. Among the different kind of predicates, we can note (see Figure 2): (1) compound terms, (2) binding terms (that is, gluing a variable to either a type expression or a compound term) and (3) constraints, which are defined by applying relational operators \(<\), \(\leq\), \(!=\), etc. among terms. An example of a binding term can be seen in line 7 of Figure 1, al is Association(name1,\_,\_,\_,\_,\_,\_), where the variable al is bound to the type expression Association. A constraint between terms is also shown in line 7, particularly in the expression al \(!=\) a2.

Rules. They are specified by the operator \(\_\_\_\), indicating, in the left hand, a simple term and, in the right hand, the list of predicates specifying the rule. A rule behaves like a universally quantified implication; whenever the relations on the right hand hold for some substitution of the variables, then the left hand holds for that same substitution [10], [11]. The intuition of rules is production; they create new entries in the fact-base of Formula, populating previous defined types with facts representing the members in the collection given in the rule.

Queries. Formula reserves a new syntax element for rules where left–hand side is a nullary construction [10], [11]. A query behaves like a propositional variable that is true if and only if the right hand side of the definition is true for some substitution [10], [11]. Queries are constructed by the operator \(\_\_\_\_\_\); and can also be used like propositional variables to construct other queries. In particular, Formula defines in every domain the conforms standard query, where all constraints come together, and which defines how a valid instance of the domain have to look like. Based on the existential quantification semantics of queries, the universal quantification can be achieved by verifying the negation of a query representing the opposite of the original predicate. For example, in order to ensure that upper bounds of associations’ multiplicities are upper than or equal to lower bounds, we firstly need to define a query representing the existence of associations verifying the opposite (see the definition of the query errorBadMultInterval in line 6 of Figure 1). With this query, we are considering such incoherent situation as a valid state. Thus, in order to verify that such situation is invalid, we include the negation (\(\_\_\_\_\_\)) of the query in the conforms query (line 9).

III. Encoding UML/OCL Models into Formula

As we have advanced previously, our proposal follows a MOF-like metamodeling approach, which is based on the framework the developers of the Formula tool give in [11]. In particular, the framework provided in [11] gives a representation in Formula of part of the key concepts defined both at the MOF meta–level [3], representing the M2 level, and at the instance–level [3], representing the M1 level for the object diagram. The resulted Formula expressions are grouped in an only Formula domain, which is used by the Formula solver to find, if it exists, a valid set of instances of arbitrary class diagrams at the M1 level (conforming with their MOF meta–level representation) and its corresponding instances at the M0 level (conforming with their instance–level representation). We remark that the authors in [10], [11] do not give a specific approach for the translation of OCL constraints.

Based on this proposal, we have extended and modified it giving weight to four main aspects. Firstly, we have mainly focused on obtaining a more faithful representation of the MOF structural distribution, specifying a richer metamodeling framework. Our extended proposal is materialized into four different Formula units distributed along the MOF meta levels, which ease the application and the understandability of our approach, while promoting units reutilization. Secondly, we provide an approach based on the CLP paradigm for analyzing model instances of specific class diagrams, and not arbitrary ones as authors in [10], [11] do, which we consider not enough when needed to reason about specific class diagrams. Thirdly, in contrast to [10], [11], we give an approach for translating OCL constraints to Formula by; (1) identifying a significantly expressive fragment of OCL, and (2) providing its translation into Formula. Finally, we have implemented part
of our translation approach based on MDA, by means of the development of our $\text{CD2Formula}$ Eclipse plug-in.

Each Formula unit defined in our approach contains two blocks of Formula expressions, related to the translation of the UML class diagram structural aspects (see Section IV) and its OCL constraints (see Section V), respectively. Our approach is illustrated with the case study of Figure 3, designed for explanation purposes covering basic aspects. In particular, this model describes both the contractual relationship between a “Company” and a “Person”, and the family recursive relationship connecting the class “Person”.

IV. TRANSLATION OF A CLASS DIAGRAM STRUCTURAL ELEMENTS

In this section, we present a brief introduction of the rules we have defined to transform a class diagram (CD), conforming with the UML metamodel [3] ($\mathcal{M}$), into Formula. Due to space reasons, in this paper we mainly focus on a set of basic structural UML class diagram features (UML class, attribute, association) for being frequently used for modeling structural aspects of systems, and also provide the translation of the UML Classifier element and Association classes. Next, we briefly explain their translation classifying the generated Formula instructions into the different MOF levels. For the explanation, we lean on Table I.

A. Classes, Associations and Properties

The translation of UML Classes, Associations and Properties into Formula follows the following proposal.

Level M0. Finally, in order that Formula can reason and search for valid instances of the specific classes, associations and properties of the source class diagram, we include the introduce ($f, n$) command (used to add $n$ terms of the element type $f$) with the corresponding $\text{InstanceName}$, Link.$name$ or $p$.name+,p.ownerName$\text{Slot}$ data type, as $f$, and a specific number as $n$, to indicate the number of valid instances of such data type we want Formula to generate as part of the resulted object class diagram. For example, we define the [introduce (InstancePerson, 2)] command, so that Formula searches two valid instances of InstancePerson (see level M0 in Table I).

B. Classifier and Association Classes

A special remark have to be made regarding the Classifier element $\in CD$ at the M2 level, and association classes $\in CD$. On the one hand, the Classifier element is defined by means of a derived data type $\epsilon \subset CD$, as the union of the Class and
Table I: Excerpt of the CD to Formula mapping.

<table>
<thead>
<tr>
<th>Level</th>
<th>Class</th>
<th>Association</th>
<th>Property</th>
</tr>
</thead>
<tbody>
<tr>
<td>M2</td>
<td>primitive Class ::= (name: String, isAbstract: Boolean)</td>
<td>primitive Association ::= (name: String, srcType: Class, srcLower: Class, srcUpper: Class, ... OCL constraints is presented in Figure 4. Firstly, the OCL expression is translated to an equivalent FOL formula.</td>
<td>primitive StrProperty ::= (name: String, def: String, lower: Natural, upper: UpperBound).</td>
</tr>
<tr>
<td></td>
<td>Translation pattern:</td>
<td>Translation pattern:</td>
<td>primitive LiteralNullProperty ::= (name: String, def: Null, p.default).</td>
</tr>
<tr>
<td></td>
<td>Class::class = Class(c.name, c.isAbstract)</td>
<td>a.memberEnd.at(1).name: Instances.memberEnd.at(1).type.name</td>
<td>primitive UnlimitedNaturalProperty ::= (name: String, def: Unbound, p.upperValue).</td>
</tr>
<tr>
<td></td>
<td>Example:</td>
<td>a.memberEnd.at(2).type.name</td>
<td>HasProperty(Class(&quot;Person&quot;, false), p.name, p.owner.isAbstract).</td>
</tr>
<tr>
<td></td>
<td>ClassPerson is Class(&quot;Person&quot;, false)</td>
<td>a.memberEnd.at(2).isAbstract</td>
<td>p.owner.name is p.type property.</td>
</tr>
<tr>
<td></td>
<td>Example:</td>
<td>a.memberEnd.at(2).isAbstract</td>
<td>HasProperty(Class(&quot;Person&quot;, false), p.owner.name, p.owner.isAbstract).</td>
</tr>
<tr>
<td>M1</td>
<td>primitive Instance::name ::= (id: Integer, type: Class)</td>
<td>primitive LinkFamily::=(id:Integer, type:Association, number).</td>
<td>p.name+p.owner.name is p.type property.</td>
</tr>
<tr>
<td></td>
<td>primitive InstancePerson::= (id: Integer, type: Class)</td>
<td>Example:</td>
<td>HasProperty(Class(&quot;Person&quot;, false), StrProperty(&quot;name&quot;, &quot;&quot;, 1, 1)).</td>
</tr>
<tr>
<td></td>
<td>Formula instructions pattern:</td>
<td>Formula instructions pattern:</td>
<td>HasProperty(Class(&quot;Person&quot;, false), StrProperty(&quot;name&quot;, &quot;&quot;, 1, 1)).</td>
</tr>
<tr>
<td></td>
<td>[Introduce(InstancePerson,2)]</td>
<td>IsInstanceMemberEnd::=(id: Integer, type: Class, isAbstract: Boolean).</td>
<td>HasProperty(Class(&quot;Person&quot;, false), p.name, p.owner.isAbstract).</td>
</tr>
<tr>
<td>M0</td>
<td>Formula instructions pattern:</td>
<td>Formula instructions pattern:</td>
<td>HasProperty(Class(&quot;Person&quot;, false), p.name, p.owner.isAbstract).</td>
</tr>
<tr>
<td></td>
<td>[Introduce(InstancePerson,2)]</td>
<td>LinkFamily::=(id: Integer, type: Association, number).</td>
<td>HasProperty(Class(&quot;Person&quot;, false), p.name, p.owner.isAbstract).</td>
</tr>
</tbody>
</table>

Association primitive data types so that we can generally refer to classes and associations. On the other hand, association classes of CD are translated in the same way as associations but with the particularity that they have associated properties. In particular, since they are translate as associations, they can register the associated classes. Additionally, in our proposal we have defined slots in such a way that their owners are of type Element (see M1b translation of properties). This Element: data type is defined as the union of Instance and Link, in such a way that we allow not only classes to have properties but also association classes.

Finally, the Formula expressions resulted from the translation of a CD are distributed into four different Formula units. On the one hand, Formula expressions defined at the meta–model level (M2) are included into a Formula domain called MetaLevelFD. Since the representation of the meta–level M2 is the same whatever CD is considered, this Formula domain is defined once and used for each CD. An excerpt of the MetaLevelFD domain has been presented in Figure 1. On the other hand, Formula expressions defined at the model level (M1) are distributed into two different units; the CDModelFM model, which is constituted by the Formula expressions defined in M1a, conforming with the MetaLevelFD domain, and the InstanceLevelFD domain, constituted by the expressions defined in M1b. Finally, the Formula expressions at the instance level (M0) are included in the CDInstanceFM partial model. Starting from these units, Formula can reason about the valid object class diagram, represented as instances of the elements of the InstanceLevelFD domain, conforming the given CD, represented by means of the CDModelFM model.

V. TRANSLATION OF CLASS DIAGRAM CONSTRAINTS

OCL integrity constraints undecidability has been tackled in the literature by defining methods that allow UML/OCL reasoning at some level. Examples of such methods are [6], [14]; (1) those that allow only specific kinds of constraints, (2) those that consider restricted models, (3) methods that do not support automatic reasoning, or (4) those that ensure only semi-decidable models. Our approach, which would fit within the first type, identifies a significantly expressive fragment of OCL and provides its translation to Formula for OCL constraints’ decidable reasoning. In this section, we show that our OCL fragment can be formally encoded in Formula, thus, we guarantee finite reasoning for every OCL CD’s constraint expressed using the constructors of our OCL fragment.

Our OCL to Formula translation relies on two foundations. Firstly, an OCL expression can be represented by means of First-Order Logic (FOL) expressions, taking into account that FOL, although less expressive than OCL, is commonly used for reasoning about the world using rules of deduction (see for example [15]). Secondly, a FOL expression can be translated into a logic constraint program P. More specifically, as stated in [16], each constraint logic program P can be translated in polynomial–time into first-order logic (FOL) according to its Clark Completion (from now on, we refer to the result of this translation as P*). Roughly speaking, the Clark Completion of a program P corresponds to the completion of every atom or predicate symbol p in P. The Clark Completion captures the reasonable assumption that the rules for each atom or predicate symbol cover all of the cases where the atom is true. Taking this into account, the Clark Completion of an atom or predicate symbol can be represented as a combination of term expressions and rules, evaluated in variables, giving a true result. The inverse translation, that is, from the FOL representation of P (P*) to P can be carried out by applying inverse versions of the Clark Completion algorithm, which compile specifications into the logic program it directly specifies, such as the one given in [17].

Based on these foundations, our proposal for the translation of OCL constraints is presented in Figure 4. Firstly, the OCL expression is translated to an equivalent FOL formula.
Figure 4: Our approach for translating OCL constraints
(see step label 1 in Figure 4). Secondly, the predicates and expressions of the resulted FOL formula are rewritten in terms of Formula elements used to represent the class diagram as stated in our approach described in Section IV, obtaining the Clark Completion version of the final Formula logic program corresponding to the OCL constraint, from now on FOL* (see step label 2 in Figure 4). Thirdly, supported by the inverse algorithm of Clark Completion, we obtain the Formula logic program (see step label 3 in Figure 4). Finally, we can use the Formula automated tool for reasoning about the Formula model with the resulted expressions. Additionally, we note that it is not necessary to have the OCL constraint initially represented using the elements included in our OCL fragment, but, when possible, we can carry out a preliminary step where such OCL expression is redefined applying OCL equivalence rules (see step label 0 in Figure 4), resulted in other OCL expression whose constructs fit within our OCL fragment.

Next, we introduce the chosen OCL fragment and go on to explain our approach for translating it. More specifically, in order that the reader can get a better idea of such translating approach, firstly we will explain the translation of a simple OCL constraint, to serve as a reference explanation for the translation of the remainder elements of our OCL fragment.

A. Introduction to the Chosen OCL Fragment

Each OCL constraint is defined in the ‘context’ of a specific instance of the corresponding UML element, reserving the ‘self’ word to refer to the instance of the classifier on which the expression is evaluated. Taking this into account, an OCL invariant has the form: context C inv: expr(self), where C is the class ∈ CD to which the invariant is applied and expr(self) is an OCL expression resulting in a Boolean value for each self ∈ C. In particular, this invariant states that, for every instance self of C existing in a system state, the property described by expr holds for self.

An OCL expression can be defined as a combination of navigation paths with OCL operations, which specify restrictions on those paths. A navigation path can be defined as a sequence of roles’ names in associations (such as p.children, being p a Person instance in Figure 3), attributes’ names (such as c.name, being c a Company instance in Figure 3), or operations (for example, c.hireEmployee(p)). Taking this into account, in Figure 5 we represent the syntax of our specific fragment, where OCLExpr is defined in a recursive manner. For example, an OCLExpr can be the result of applying relational operations to AddExpr expressions. Additionally, an OCLExpr can be the result of applying a boolean operation BoolOper to a Path, or a Path to which a SelectExpr is applied. An OCLExpr can be also constituted by boolean combinations of these OCL expressions (not, and, or). A Path expression represents the structural way of defining navigation paths, starting from a

Figure 5: Syntax of the OCL fragment.

PathItem, by combining roles’ names, attributes’ names or operations, with the dot operator. For an explanation of OCL, we refer to [4].

B. Our Translation Approach

Formula does not have a concept similar to that of OCL invariants but gives the possibility of defining queries, which provide a way to represent invariant semantics. As way of example of our approach, in this section we introduce the basic rule for translating OCL invariants where the OCLExpr corresponds to a simple relational expression RelExpr. We explain this rule by applying it to the invariant context Person inv: self.age >=18, which formalizes the constraint “The people registered in the system must be older than 18 years old” (see Table II).

First-step. This step is carried out by means of an interpretation function FOL(), which translates each OCL expression expr(self) defined in an instance self ∈ C, into a First–Order Logic (FOL) formula defined in the variable self (see label (1) in the first step of Table II). Basis in first order logic states that the universal quantifier corresponds to a negated existential, so the previous expression is equivalent to the one label (1'), where FOL(not expr(self)), corresponds to the mapping of not expr(self) into First–Order Logic.

Second-step. As described previously, each constraint logic program P can be translated into first-order logic (FOL) according to its Clark Completion P* [16]. Roughly speaking, the Clark Completion of an atom or predicate symbol can be represented as a combination of term expressions and rules, evaluated in variables, giving a true result.

Taking this into account, the second step is devoted to represent the semantics given by the affirmative evaluation of FOL(not expr(self)) in the collection of instances self ∈ C, by means of Formula expressions. Since paths in OCL are defined in terms of instances of the class diagram, and in our approach such instances are defined by means of the data types defined in the CDInstancePM partial model, such Formula expressions are written in terms of the InstanceClassName+LinkassociationName and/or propertyName+ownerNameSlot data types. Based on this premise, in this second step we rewrite the FOL expression FOL(not expr(self)) in terms of Formula expressions by applying a second function called FOL*(). This function FOL*() basically represents the predicate FOL(not expr(self)) by using the corresponding Formula terms and predicate symbols ∈ InstanceLevelFD, and Formula constraints, in such a way that...
Table. II: Translation of an invariant and example of use.

<table>
<thead>
<tr>
<th>OCL Invariant</th>
<th>Translation Approach</th>
</tr>
</thead>
<tbody>
<tr>
<td>self.age &gt;= 18</td>
<td>$\text{C}\rightarrow\text{size}()$ $\text{query}=\text{CLP}^{*}(\text{FOL}(\text{not}\text{exp}(\text{c})))$. (3)</td>
</tr>
<tr>
<td>$\text{C}\rightarrow\text{select}(\text{c}\text{exp}(\text{c}))$ $\text{SCexpType}(\text{self},\text{sele}):=\text{CLP}^{*}(\text{FOL}(\text{exp}(\text{c})))$</td>
<td></td>
</tr>
<tr>
<td>$\text{C}\rightarrow\text{forAll}(\text{c}\text{exp}(\text{c}))$ $\text{query}=\text{CLP}^{*}(\text{FOL}(\text{not}\text{exp}(\text{c})))$. conform:= ! query.</td>
<td></td>
</tr>
<tr>
<td>$\text{C}\rightarrow\text{forAll}(\text{c}\text{exp}(\text{c}))$ $\text{query}=\text{CLP}^{*}(\text{FOL}(\text{not}\text{exp}(\text{c})))$. conform:= ! query.</td>
<td></td>
</tr>
</tbody>
</table>

The resulted expression is evaluated to true (see step labeled (2) in Table II). In particular, the application of this step to our constraint consists of representing age(self)<18 in terms of the agePersonSlot whose val property is less than 18.

Third-step. Taking into account the semantics of queries in Formula, the FOL expression given in the second step is finally represented by means of the definition of a query and the verification of its negation in the conforms query (see step labeled (3) in Table II). This step is materialized by means of the application of the function $\text{CLP}^{*}$, which basically rewrites the terms resulted from (2), and joins them by ".

To sum up, the translation of an invariant $I$ is carried out by means of the composition of the three functions, $\text{CLP}\circ\text{FOL}^{*}\circ\text{FOL}()$.

C. Translation Approach of More Complex OCL Expressions

Having presented our approach for the translation of a simple OCL invariant, next we make some remarks regarding the translation of the remainder elements in our OCL fragment. More specifically, in Table III we present the translation rules we define for the conjunction, disjunction and the negation operators considered in the OCL fragment. For example, we describe the translation of an OCL expression with the conjunction operator $E_1$ and $E_2$ as: $\text{CLP}^{*}(\text{FOL}(\text{E}_1)) \& \text{CLP}^{*}(\text{FOL}(\text{E}_2))$, where each expression is translated recursively using the translation rules presented in the rest of this paper by applying the defined functions. In particular, if $\text{CLP}^{*}(\text{FOL}(\text{E}_1))$ results in the verification of a query $\neg\text{query}_1$ in the conforms one, and $\text{CLP}^{*}(\text{FOL}(\text{E}_2))$ results in the verification of another query $\neg\text{query}_2$, the result of translating the conjunction is the expression $\neg\text{query}_1 \& \neg\text{query}_2$ specified in the conforms query (that is, conforms:= $\neg\text{query}_1 \& \neg\text{query}_2$). The translation of the disjunction operator, on the other hand, results in the expression conforms:= $\neg\text{query}_1 \lor \neg\text{query}_2$, being $\neg\text{query}_1$ and $\neg\text{query}_2$ the translations of $E_1$ and $\text{CLP}^{*}(\text{FOL}(\text{E}_2))$, respectively. Finally, the translation of the negation operator results in the expression conforms:= $\neg$ query, being $\neg$ query the translation of not E.

The remainder OCL expressions in our framework include operations in collections. Selecting the exclusion and transitive closure elements, whose translation requires extra attention, we consider that the translation of the remainder OCL elements (forAll [4] (p. 29, Section 7.7.3) and size [4] (p. 157, Sec. 11.7.1)) can be easily understood by considering our previous explanations. Next, we briefly describe our approach for their translation into Formula.

Select operation. Since this operation refers to obtaining a subcollection from a set of elements ([4] pp. 27, Sec. 7.1.1), its translation consists of defining a new Formula data type and populate it with the facts representing the members in the collection we want to select (see the first and second lines, respectively, of the translation of the select operation in Table III). As a way of example, if we want to collect the female employees of a company, we define the type: FemaleEmp::= (self: InstanceCompany, sele:InstanceEmployee), and populate it by means of the following rule, which gathers only female employees:

FemaleEmp(self,sele) :-
    LinkContract(_,_,sele,self),
    genderPersonSlot(sele,_,val),
    val=female.

Transitive closure. Transitive closure is normally needed to represent model properties which are defined in a recursively fashion. The translation of closures is not straightforward since they are not finitely axiomatizable in first order logic, and OCL also does not support them natively [18]. Nevertheless, it is possible to define the transitive closure of relations that are known to be finite and acyclic. In particular, for its translation we have based on both, the definition of transitive closure provided in [18], and the representation in CLP of acyclicity constraints provided in [11] (p. 3), and proposed a translation based on defining Formula rules, considering the fact that CLP exposes fixpoint operators via recursive rules. Additionally, the translation of this operation allows us to support aggregation.

Finally, the Formula model resulted from the translation of a class diagram model annotated with OCL constraints (that is, the 4 Formula units including the Formula translation of the OCL constraints), is used by Formula for reasoning about it. More specifically, the tool inspects the Formula model looking for a valid and non-empty instantiation of the CD/OCL model to proof its satisfiability. If the result is empty, the defined CD/OCL model is not satisfiable. Otherwise, Formula proposes a conforming instantiation model of the defined CD/OCL model, according to the desired software system settings.
VI. AUTOMATIC TRANSLATION

In order to manually transform a class diagram into the Formula language, a professional with both UML and Formula skills may be required. Additionally, such an encoding process may entail a big effort depending on the class diagram used. The challenge is to perform such a transformation in a viable and cost-effective way. The complexity of some software designed models together with their possibility of change over time, make the manual transformation of every class diagram representing a software model into the input language of a model finder tool, a cumbersome and costly endeavor. To overcome these challenges, we have based on an MDA tool-approach to automatically carry out the translation of a class diagram to Formula. More specifically, we have developed an Eclipse plug-in, called CD2Formula plug-in, which gives support for the class diagram to Formula transformation as stated in our proposal (see in Figure 6 a snapshot of the plug-in). The idea is that the defined plugin together with the Formula tool, constitute the complete proposed framework for class diagram to Formula specification. Firstly, by means of the CD2Formula plugin we automatically generate, from a class diagram, the Formula specification, which is taken by the Formula model finder for reasoning about the input class diagram model.

The core of our plug-in is that it itself uses a MDA-based plug-in that gives support for customizable model-to-text (M2T) transformations. Among the large amount of MDA-based tools in the literature, we have chosen the MOFScript Eclipse plug-in, which we have already used in previous works [19], [20]. MOFScript is an Eclipse plug-in [21], [22] that implements the MOFScript language, which was one of the candidates in the OMG RFP process on MOF Model-to-Text Transformation [23]. As input models, MOFScript can use any model that complies with the EMF [24] metamodel. From these input models, the tool can generate any arbitrary text (such as Java code or XML) by using a defined set of MOFScript transformations. Each MOFScript transformation consists of transformation rules that are basically the same as functions, and which define the behavior of the transformation. The transformation rules are defined based on the metamodel and subsequently compiled and executed on the model.

In our particular case, we use the UML 2.0 metamodel and a class diagram that represents the software design as the model. To create class diagram models, we can use any UML 2.0 compliant tool that can create models, as .uml extension files, in the XMI format supported by EMF (e.g., the UML2 Eclipse plug-in [25]).

As far as the Formula program generation is concerned, an important remark must be made. In our proposal for the Formula representation of a UML class diagram, we need to include specific Formula instructions to tell the Formula solver the number of valid instances (for example, the number of class and association instances), we would like for the final solution. Such number of instances is set by means of the Introduce Formula instructions, which, in our particular case, are included in the CDInstance FPM partial model defined for each class diagram. Since such number of instances should be established by the user before carrying out the transformation from the class diagram to the Formula specification, we firstly need to ask the user for such information, which is specific for each class diagram. Taking this into account, we have defined two sets of MOFScript transformations, devoted respectively to: (1) generate a java GUI (Graphical User Interface), which ask the user for the required information, and (2) create the Formula specification for the class diagram (whose CDInstance FPM partial model is generated taking into account the values inserted by the user by means of the previously generated GUI interface). Both sets of MOFScript transformation files are devoted to produce the print statements that generate the java GUI interface and the different Formula units, respectively.

Particularly, in the definition of the MOFScript transformations, we have followed a concrete idea, which consists on defining two kinds of transformation rules: (1) those that tra-
verse the model and collect the information in it and (2) those
that generate actual code (java print statements or Formula
structures, respectively). The first kind gathers data and records
them in collections (such as lists or hashtables) or other built-
in types. Finally, the code generation rules use this information
in print statements. As a way of example, an extract of one of
the defined rules is shown in Figure 7. In particular, this rule
called CDtoCDModel will create the Formula expressions that
constitute the $CDModel_{FM}$ model (such as classPerson
is Class('Person', 'false')).

Regarding the MOFScript transformation files defined
to generate the Formula units, we have created 3 files:
main.m2t, helpers.m2t, and FormulaUnits.m2t.
The transformation file main.m2t contains the main rule
that actually generates the complete Formula specification of
the class diagram by using specific rules from the rest of the
defined transformation files. The file helpers.m2t has been
defined to be used as a library, containing commonly used
rules that are required by other rules during the transformation
process. Finally, the FormulaUnits.m2t file contains the
rules devoted to finally produce the print Formula structures
that constitute the three Formula units in our approach, which
depend on the specific class diagram.

As for the generation of the GUI interface for a specific
class diagram, we have defined an only MOFScript transforma-
tion file called main.m2t. This file defines MOFScript
rules that mainly traverse the class diagram and generate the
java print statements that define the different labels and
text fields of the form, together with a button to send
the inserted data (see the GUI interface CD2Formula labeled
2 in Figure 6) to be used for the second transformation.
In particular, the second group of MOFScript files gets the
data given by the user through the GUI interface, thanks to
a specific MOFScript's functionality. MOFScript allows the
possibility of invoking java methods from MOFScript rules and
retrieving the returned information. Taking this functionality
into account, the java GUI interface file contains a specific java
method in such a way that, when the form in the java GUI
interface is filled out, such method returns a hashtable with the
pairs (type of instance–number of instances desired). In
this way, the second group of MOFScript transformation files
takes such information to print the corresponding Introduce
Formula instructions of the final Formula specification.

We want to highlight that since the MOFScript transfor-
mation rules are defined based on our transformation rules
between a class diagram and the Formula model, and these
transformation rules are defined independently of the class
diagram used, the MOFScript transformations do not have to
be modified to translate a different class diagram.

Having defined the two sets of MOFScript transforma-
tion files, we have developed an Eclipse plug-in called
CD2Formula in such a way that it integrates such MOFScript
rules so that the transformation from a class diagram to
its Formula specification can be generated in an automatic
fashion. More specifically, the plug-in provides a menu option
available for each UML class diagram (specified as .uml
extension files), which allows the execution of the MOF-
Script transformations. The transformation process encom-
passes three steps. Firstly, the user chooses the menu option
the plug–in provides, which executes the first set of MOFScript
transformations that lead to the dynamic creation of the GUI
interface. Secondly, the plug–in refreshes the Eclipse project
so that the corresponding interface java class can be created
and instantiated. Thirdly, the second set of MOFScript trans-
formations is executed, which (1) leads to the invocation of the
java method that shows the interface, asking the user for the
required values, (2) retrieves the values inserted by the user in
the interface, and (3) generates the Formula specification (that
is, the FormulaSpecifications.4ml file), using such
values. Finally, the resulted file is used by the Formula tool
for reasoning about the class diagram.
About the usability of the proposal and the developed plug-in, we have to say that it is only required a professional with OCL skills in order to be able to apply our OCL to Formula translation proposal to the OCL constraints defined in the specific class diagram. Excluding it, no specific knowledge would be required for managing the plug-in since its interface has been developed so that it is simple and easy to use.

VII. DISCUSSION AND RELATED WORK

As described previously, the formalization and analysis of UML class diagrams can be done by means of translating the model to other language that preserves its semantics, and finally, using the resulted translation to reason about the design. Taking into account that there is not an only language for materializing such translation, and that several translation approaches can be established using a same language, a discussion about the semantic support of the language, together with the strengths and weaknesses of the particular translation approach, is worthwhile. Our work bets on using Formula for the semantics preserving translation of the models to be verified. As for the use of Formula instead of other analyzers, in particular, Formula authors present in [11] a comparison with other tools, both SAT (Boolean Satisfiability) solvers and alternatives such as ECLiPSE and UMLtoCSP, focusing mainly on Alloy [26], for being the closest tool to Formula. Although the Formula authors provide a careful comparison with Alloy in [11], it is worth noting the strengths of Formula, such as a more expressive language or its model finding problems, which are in general undecidable.

Our approach follows a multilevel MOF-like framework based on the one proposed in [11]. On the one hand, we propose a more faithful representation of the basic UML metamodel and instance domain elements [3]. We consider that providing a translation that captures the structural distribution of the MOF architecture can contribute to ease the application and understandability of the representation of a CD/OCL model into Formula. We also give support for the translation of more metamodel elements (such as full support to generalization, property types other than Integer, String and Boolean, including user defined data types, property’s multiplicities, etc.), thus providing a richer framework. Additionally, we enhance the proposal given in [11] by identifying an expressive fragment of OCL, which guarantees finite satisfiability and providing a formalization of the transformations from such OCL fragment to Formula. At this respect, several related works can be cited, being one of the most complete proposals the one given in [14]. In [14], the authors define a fragment of OCL called OCL–lite, and prove the encoding of such a fragment in the description logic ALECT, so that Description Logic techniques and tools can be used to reason about class diagrams annotated with OCL–lite constraints. A difference of this approach with ours is the fact that, although the chosen fragment is quite similar than ours, we have tried to identify a simplest fragment so that no element included in it can be inferred from other constructors in the fragment by applying direct OCL equivalences (such as the implies operator). In our particular case, there are several OCL operations and expressions whose representation in Formula is straightforward by applying equivalences (such as the exists operations) [4] (p. 30, Sec. 7.7.4). isEmpty/notEmpty [4] (p. 157, Sec. 11.7.1), xor [4] (p. 153, Sec. 11.5.4), or reject [4] (p. 27, Sec. 7.7.1).

On the other hand, there are other elements, such as oclIsTypeOf, which is considered in the OCL–lite fragment but that can not be represented into formula. More specifically, Formula does not support the translation of, for example, the following OCL properties [4] (p. 146, Sec. 11.3): (1) oclIsTypeOf(t : OclType), which is used to know whether the object to which it is applied is of type t, and (2) oclIsKindOf(t : OclType), which returns true whether t is either the direct type of the object to which the operation is applied or a supertype of the object. As for the representation in Formula of the oclIsTypeOf operation, there is no way to know the type of a variable by using the Formula syntax, but the mismatch among variables and types is verified by the Formula checker. The same argument is applied to the OclIsKindOf operation. Similarly happens with OCL operations that are state dependent (such as the operations oclIsInState(t: OclType), which evaluates whether the object is in a specific state, and oclIsNew(t: OclType), which checks whether the object does not exist in the previous state of the system but exists in the current state). In both cases, a UML statemachine diagram is required, and although representing UML statemachines in Formula could constitute an interesting issue for future work in order to give support to reason also about dynamic system models, it is out of the scope of this work. Focusing on reasoning about static class diagrams models, in spite of these operators, we give support to other not straightforward operators, such as transitive closure, not normally included in related works.

VIII. CONCLUSION AND FUTURE WORK

We present an overall framework to reason about UML/OCL models based on the CLP paradigm, using Formula. Our framework provides a way to translate a UML model into Formula, following a MOF-like approach. We also identify an expressive fragment of OCL, which guarantees finite satisfiability and we provide an approach for translating it to Formula. We also provide an implementation of our UML to Formula proposal by the development, following a Model Driven Architecture (MDA) approach, of the CD2Formula plug-in. Particularly, starting from a UML class diagram representing the static structure of a software system, our plug-in carries out the automatic generation of the Formula specification corresponding to such UML model, by simply choosing a menu option the plug-in provides. The proposed framework can be used to reason, validate and verify UML software designs by checking correctness properties and generating model instances using the model exploration tool Formula.

Although we support the automatic translation from a UML class diagram to Formula by means of our plug-in, the automatic translation to Formula of specific class diagram’s OCL constraints specified using our OCL fragment constitutes a remaining work.

ACKNOWLEDGMENTS

This work has been partially supported by the Academy of Finland, the Spanish Ministry of Science and Innovation (project TIN2009-13584), and the University of La Rioja (project PROFAI13/13).
REFERENCES


