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

Beatriz Pérez
Department of Mathematics and Computer Science,
University of La Rioja,
Logroño, Spain.
Email: beatriz.perez@unirioja.es

Ivan Porres
Department of Information Technologies,
Åbo 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. Additionally, 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 UserDataTypes {
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 errorBadMultInterval := Association(_,_,srcLower,srcUpper,_,_,_),
                                          srcLower >srcUpper.
7 errorMetaDupAssoc := a1 is Association(name1,_,_,_,_,_,_),
                       a2 is Association( name2,_,_,_,_,_,_),
                       name1 = name2, a1 != a2.
8 ...
9 conforms := !errorBadMultInterval & !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 `name1` 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

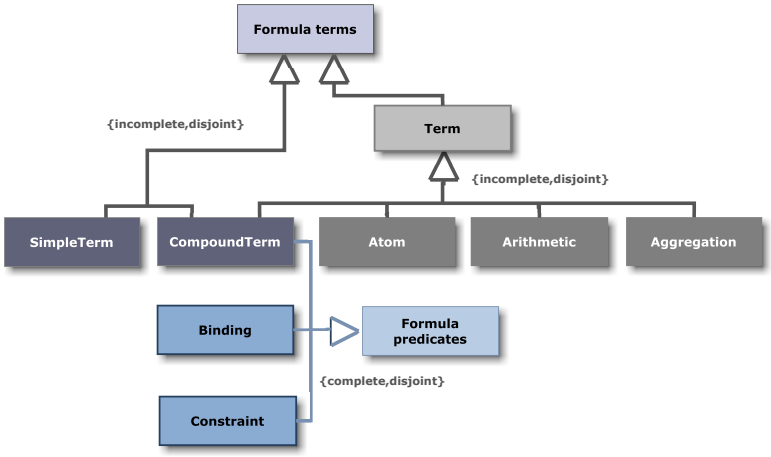

Figure. 2: Several of the Formula structural elements.

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 <,>=, !=, etc. among *terms*. An example of a binding term can be seen in line 7 of Figure 1, `a1 is Association(name1,_,_,_,_,_)`, where the variable `a1` is bound to the type expression `Association`. A constraint between terms is also shown in line 7, particularly in the expression `a1 != 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 be also 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

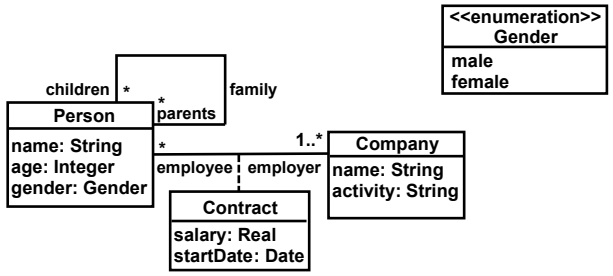

Figure. 3: Case study.

of our translation approach based on MDA, by means of the development of our *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 ($\mathcal{CD}$), 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 M2. For each meta model element *Class*, *Association* or *Property* ϵ $\mathcal{M}$, we define a primitive Formula data type with the same name and with specific *fields* (see level M2 in Table I). For example, in the case of classes, we define the data type `Class(;)` ϵ *CPS*, with two `String` fields (`name` and `isAbstract`). The definition of these data types allows Formula to create Formula instances representing specific UML classes, associations and types of properties, respectively, at the M1 level. In the case of the *Property* element ϵ $\mathcal{M}$, we define a type for each *build-in type*, called *typeName*`Property`, with specific fields (see Table I). In addition to `Integer`, `String` and `Boolean`, included in [11], we also give support to `Real`, `LiteralNull` and `UnlimitedNatural` types. The data type `HasProperty(;)` ϵ *CPS* is also defined to represent the possession of a property by a classifier.

Level M1. Two groups of expressions are defined at this level. **[M1a.]** Each specific *class*, *association* and *property* ϵ $\mathcal{CD}$, is represented by a Formula instance of the corresponding constructor (`Class`, `Association` or `Property` ϵ *CPS* defined at level M2). By these Formula instances, we are explicitly representing, in contrast to [10], [11], not arbitrary classes in a class diagram but specific ones. For example, the elements `ClassPerson` and `family` defined in `M1a` of Table I correspond to two Formula instances of the constructor `Class` and `Association`, respectively, defined at M2. In particular, specific properties ϵ $\mathcal{CD}$ are represented by a Formula instance of the corresponding `Property` constructor (e.g., *namePersonP is Str*`Property(...)` in `M1a` of Table I), and by an instance of the data type `HasProperty` ϵ *CPS*, representing the property's ownership (see Table I).

[M1b.] In order that Formula is able to generate instances of specific *class*, *association* and *property* ϵ $\mathcal{CD}$ to explore the concrete design–space, we need to create specific Formula data types representing each type of instance. For their definition, we have based on the description of the *Instances* package [3], in particular, on the *InstanceSpecification* element, for classes and associations, and on the *Slot* element, for properties. On the one hand, the definition of the UML *InstanceSpecification* element includes the classifier of the represented instance and the associated *InstanceValue* [3]. Taking this into account, for each *class c* ϵ $\mathcal{CD}$, we define a primitive Formula data type called `Instance`*c.name*`(;)` ϵ *CPS*, with two fields, representing the associated classifier and instance value, respectively (see level M1b in Table I). As a way of example, see the primitive data type `InstancePerson` in Table I. When the classifier is an association, the UML instance specification describes a *link* [3], so in this situations we name the created data types with the `Link` prefix. Since links connect class instances [3], for each *association a* ϵ $\mathcal{CD}$, we define a primitive Formula data type called `Link`*a.name*`(;;;)` ϵ *CPS*, which includes, additionally, the instance specifications of the associated classes (see for example `LinkFamily` in Table I). So that Formula can generate property's specific values, we define specific data types representing the property's slots, based on the specifications of the *Slot* element [3]. Taking this into account, for each *property* ϵ $\mathcal{CD}$, we define a primitive type called *p.name+p.owner.name*`Slot(;;)` ϵ *CPS* (e.g., `namePersonSlot` in Table I), which registers the owner, the property type and its value.

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 `Instance`*c.name*, `Link`*a.name* or *p.name+p.owner.name*`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 ϵ $\mathcal{CD}$ at the M2 level, and *association classes* ϵ $\mathcal{CD}$. On the one hand, the *Classifier* element is defined by means of a derived data type ϵ $\amalg \subset$ *CPS*, as the union of the `Class` and

Table. I: Excerpt of the CD to Formula mapping.

Level		Class	Association	Property
M2		**primitive Class ::= (name: String, isAbstract: Boolean).**	**primitive Association ::= (name: String, srcType: Class, srcLower: Natural, srcUpper: UpperBound, dstType: Class, dstLower: Natural, dstUpper: UpperBound).**	**primitive StrProperty::=(name:String, def:String, lower:Natural, upper:UpperBound).** ... **primitive LiteralNullProperty::=(name: String, def: Null,...). primitive UnlimitedNaturalProperty::=(name:String, def: UpperBound,.) Property::= StrProperty + ...+ userDataTypeProperties. primitive HasProperty ::= (owner: Classifier, prop: Property).**
M1	a	*Translation pattern:* **Class***c.name* **is Class(***"c.name", c.isAbstract***)** *Example:* **ClassPerson is Class("Person", false)**	*Translation pattern:* *a.name* **is Association(***"a.name",* **Class("***a.memberEnd.at(1).type.name***",** *a.memberEnd.at(1).type.isAbstract a.memberEnd.at(1).lowerValue, a.memberEnd.at(1).upperValue,* **Class("***a.memberEnd.at(2).type.name***",** *a.memberEnd.at(2).type.isAbstract a.memberEnd.at(2).lowerValue, a.memberEnd.at(2).upperValue)* Example: **family is Association("family",Class("Person",false), 0, 2, Class("Person",false), 0, star)**	*Translation pattern:* *p.name+p.owner.name***P is** *p.type***Property(***"p.name",p.default, p.lowerValue,p.upperValue)* **HasProperty(Class("***p.owner,..name***",** *p.owner.isAbstract),* *p.type***Property(***"p.name",p.default, p.lowerValue,p.upperValue)***)* *Example:* **namePersonP is StrProperty("name","",1,1) HasProperty(Class("Person",false),StrProperty("name","",1,1))**
	b	*Translation pattern:* **primitive Instance***c.name* **::= (id: Integer, type: Class).** *Example:* **primitive InstancePerson::=(id: Integer, type: Class).**	*Translation pattern:* **primitive Link***a.name* **::=(id: Integer, type: Association,** a.memberEnd.at(1).name: **Instance**a.memberEnd.at(1).type.name, a.memberEnd.at(2).name: **Instance**a.memberEnd.at(2).type.name). Example: **primitive LinkFamily::=(id:Integer,type:Association, child:InstancePerson, parent:InstancePerson).**	*Translation pattern:* **primitive** *p.name+p.owner.name***Slot ::= (owner:Element, prop:***p.type***Property, value***: valueType***)** *Example:* **primitive namePersonSlot::= (owner: Element, prop: StrProperty, value:String).**
M0		*Formula instructions pattern:* **[Introduce(Instance***c.name, number* **)]** *Example:* **[Introduce(InstancePerson,2)]** *Example of the Formula generated instances:* **InstancePerson(93,Class("Person",false)) InstancePerson(96,Class("Person",false))**	*Formula instructions pattern:* **[Introduce(Link**a.name, *number* **)]** *Example:* **[Introduce(LinkFamily,2)]** *Example of the Formula generated instances:* **LinkFamily(5, Association("family",Class("Person",false),0,2, Class("Person",false),0,star), InstancePerson(93, Class("Person",false)), InstancePerson(96, Class("Person",false)))**	*Formula instructions pattern:* **[Introduce(***p.name+p.owner.name***Slot***, number* **)]** *Example:* **[Introduce(namePersonSlot,2)]** *Example of the Formula generated instances:* **namePersonSlot(InstancePerson(93,Class("Person",false)), StrProperty("name","",1,1),202) namePersonSlot(InstancePerson(96,Class("Person",false)), StrProperty("name","",1,1),201)**

`Association` primitive data types so that we can generally refer to classes and associations. On the other hand, *association classes* ϵ $\mathcal{CD}$ are translated in the same way than associations but with the particularity of that they can 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 $\mathcal{CD}$ 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 $MetaLevel_{FD}$. Since the representation of the meta–level M2 is the same whatever $\mathcal{CD}$ is considered, this Formula domain is defined once and used for each $\mathcal{CD}$. An excerpt of the $MetaLevel_{FD}$ 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 $CDModel_{FM}$ model, which is constituted by the Formula expressions defined in `M1a`, conforming with the $MetaLevel_{FD}$ domain, and the $InstanceLevel_{FD}$ domain, constituted by the expressions defined in `M1b`. Finally, the Formula expressions at the *instance level* (M0) are included in the $CDInstance_{FPM}$ partial model. Starting from these units, Formula can reason about the valid object class diagram, represented as instances of the elements of the $InstanceLevel_{FD}$ domain, conforming the given $\mathcal{CD}$, represented by means of the $CDModel_{FM}$ 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

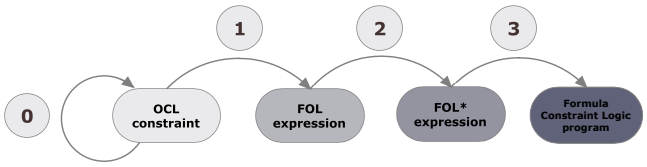

Figure. 4: Our approach for translating OCL constraints

(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 *I* has the form: `context C inv: expr(self)`, where `C` is the class $\epsilon\ \mathcal{CD}$ to which the invariant is applied and `expr(self)` is an OCL expression resulting in a `Boolean` value for each `self` ϵ `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` and `or`). A `Path` expression represents the structural way of defining navigation paths, starting from a

```
OCLExpr    ≡RelExpr |Path BoolOper |Path SelectExpr
            not OCLExpr | OCLExpr1 and OCLExpr2
            OCLExpr1 or OCLExpr2
Path       ≡PathItem | PathItem.Path
PathItem   ≡role | classAttr | operation
            roleName.role | roleName.classAttr
            roleName.oper | roleName.transClosuOper
RelExpr    ≡AddExpr <,<=,>,>=,=,!= AddExpr
AddExpr    ≡MulExpr | AddExpr +/- AddExpr
MulExpr    ≡Path | MulExpr * Path | MulExpr/Path
SelectExpr ≡-> select( OCLExpr) BoolOp|
            -> select( OCLExpr) SelectExpr
BoolOper   ≡  -> size()| ->  forAll(OCLExpr)
```

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` ϵ `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` $\in$ `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 $CDInstance_{FPM}$ partial model, such Formula expressions are written in terms of the `Instance`*className*, `Link`*associationName* and/or *propertyName*+*ownerName*`Slot` 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 $\in$ $InstanceLevel_{FD}$, and Formula constraints, in such a way that

Table. II: Translation of an invariant and example of use.

Translation of a `RelExpr` invariant	
OCL Invariant: `context C inv: expr(self)`	
First–step:	$\forall$`self` $\in$ `C` *FOL*`(expr(self)).` (1) $\neg(\exists$`self` $\in$ `C` *FOL*`(not expr(self)).` (1')
Second–step:	$\neg$(*FOL**`(C)` *FOL**`[`*FOL*`(not expr(self))]`) (2)
Third–step:	`query:=`*CLP*(*FOL**`[`*FOL*`(not expr(self))]`) `conforms := ! query.` (3)
Example of application	
OCL Invariant: `context Person inv: self.age >=18`	
First–step:	$\forall$`self` $\in$ `Person age(self)>=18.` (1) $\neg(\exists$`self` $\in$ `Person age(self)<18).` (1')
Second–step:	$\neg(\exists$`self` $\in$ `InstancePerson(id,type)` `agePersonSlot(self,def,val)` `val<18)`.(2)

Third–step:	`query:=agePersonSlot(self,_,val),`
	`val<18.`
	`conforms := ! query.` (3)

Table. III: Translation of part of our OCL fragment.

OCL expression	Translation approach
`E1 and E2`	*CLP*(*FOL**(*FOL*`(E1)`))&*CLP*(*FOL**(*FOL*`(E2)`))
`E1 or E2`	*CLP*(*FOL**(*FOL*`(E1)`))|*CLP*(*FOL**(*FOL*`(E2)`))
`not E`	*CLP*(*FOL**(*FOL*`(not E)`))
`C-> size()`	`count(`*CLP*(*FOL**(*FOL*`(C)`))`.`
`C-> forAll(c|exp(c)`	`query:=`*CLP*(*FOL**(*FOL*`(not exp(c))`)). `conforms:= ! query.`
`C-> select(c|exp(c)`	$S_{C,expr}$`Type::=(self:`T_{self}`,sele:`T_{sele}) $S_{C,expr}$`Type(self,sele):-` *CLP*(*FOL**(*FOL*`(exp(c))`))

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 *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, *CLP* $\circ$ *FOL** $\circ$ *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 `E1 and E2` as: *CLP*(*FOL**(*FOL*`(E1)`)) & *CLP*(*FOL**(*FOL*`(E2)`)), 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 *CLP*(*FOL**(*FOL*`(E1)`)) results in the verification of a query `!query1` in the `conforms` one, and *CLP*(*FOL**(*FOL*`(E2)`)) results in the verification of another query `!query2`, the result of translating the conjunction is the expression `!query1 & !query2` specified in the `conforms` query (that is, `conforms:= !query1 & !query2`). The translation of the *disjunction* operator, on the other hand, results in the expression `conforms:= !query1 | !query2`, being `!query1` and `!query2` the translations of `E1` and *CLP*(*FOL**(*FOL*`(E2)`)), respectively. Finally, the translation of the *negation* operator results in the expression `conforms:= !query`, being `!query` the translation of `not E1`.

The remainder OCL expressions in our framework include operations in collections. Excluding the `select` 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.

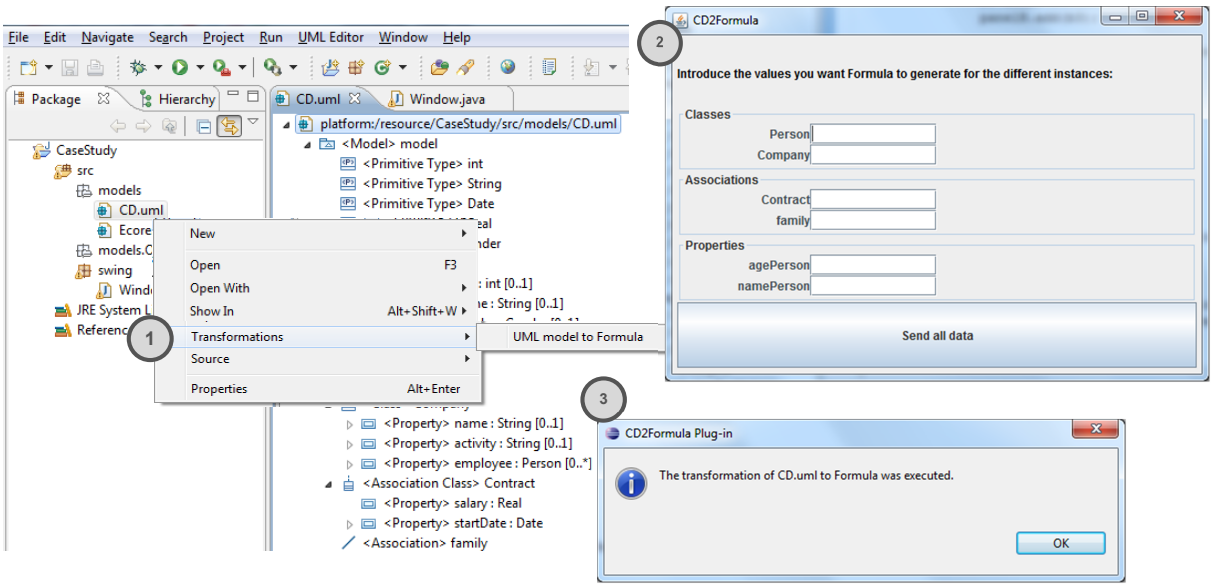

Figure. 6: A snapshot of the *CD2Formula* plug–in.

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-

```
module::CDToCDModel(){
   standardClassDiagram()
   var i:integer
   var upper:String
   var propertyType: String
   //Create an instance of the Class element for each class in the CD
   classes -> forEach(c:uml.Class){
     println('   class'+c.name+' is Class("'+c.name+'", '+c.isAbstract+')')
   }
   //Create an instance of the Association element for each association in the CD
   associations -> forEach(a:uml.Association){
     i=0
     println('   '+a.name+' is Association("'+a.name+'", ')
     print('      ')
     a.memberEnd -> forEach(p:uml.Property){
         i=i+1
         if(p.upper==-1)
            upper='star'
         else
            upper=p.upper
         print('class'+p.type.name+', '+p.lower+', '+ upper)
         if(i==1)
             print(', ')
      }
     println(')')
   }
   //Create an instance of the Property element for each property in the CD
   ...
   println('}')
}
```

Figure. 7: An extract of a Mofscript rule.

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 transformation 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 transformation 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 transformation 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 MOFScript transformations. The transformation process encompasses 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 transformations 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 $\mathcal{ALCI}$, 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` [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

[1] B. Pérez and I. Porres, "Reasoning about UML/OCL models using constraint logic programming and MDA," Proceedings of the 8th International Conference on Software Engineering Advances (ICSEA'13), 2013, pp. 228–233.

[2] J. Bézivin,"Model driven engineering: an emerging technical space," Proceedings of the International Summer School of Generative and Transformational Techniques in Software Engineering (GTTSE'05), 2006, LNCS, vol. 4143, pp. 36–64.

[3] OMG, UML 2.4.1 Superstructure Specification, Document formal/2011-08-06. Available at: http://www.omg.org/. Last visited on May 2014.

[4] OMG, Object Constraint Language, Version 2.3.1, OMG Document Number: formal/2012-01-01. Available at: http://www.omg.org/spec/OCL/2.3.1/PDF. Last visited on May 2014.

[5] A. Calì, D. Calvanese, G. De Giacomo, and M. Lenzerini, "A formal framework for reasoning on UML class diagrams,", Proceedings of the 13th International Symposium of Foundations of Intelligent Systems (ISMIS'02), LNCS, vol. 2366, 2002, pp. 503-513.

[6] J. Cabot, R. Clarisó, and D. Riera, "Verification of UML/OCL class diagrams using constraint programming," Proceedings of the 2008 IEEE International Conference on Software Testing Verification and Validation Workshop (ICSTW'08), IEEE Computer Society, 2008, pp. 73–80.

[7] V. Del Bianco, L. Lavazza, and M. Mauri, "Model checking UML specifications of real time software," Proceedings of the 8th International Conference on Engineering of Complex Computer Systems (ICECCS 2002), IEEE Computer Society, Los Alamitos, 2002, pp.203-.

[8] V. Del Bianco, L. Lavazza, and M. Mauri, "A formalization of UML statecharts for real-time software modeling," The 6th Biennial World Conference On Integrated Design Process Technology (IDPT 2002), "Towards a rigorous UML" session, Pasadena. 2002.

[9] Formula - Modeling Foundations, Website: http://research.microsoft.com/en-us/projects/formula. Last visited on May 2014.

[10] E. K. Jackson, T. Levendovszky, and D. Balasubramanian, "Reasoning about metamodeling with formal specifications and automatic proofs," Proceedings of the 14th International Conference of Model Driven Engineering Languages and Systems (MODELS 2011), 2011, pp. 653-667.

[11] E. K. Jackson, T. Levendovszky, and D. Balasubramanian, "Automatically reasoning about metamodeling," Software & Systems Modeling, February, 2013, doi:10.1007/s10270-013-0315-y.

[12] OMG, OMG Model Driven Architecture, Document omg/2003-06-01, 2003, Available at: http://www.omg.org/. Last visited on May 2014.

[13] Formula 1.3. Formula documentation (Help). Formula downloads, available at: http://research.microsoft.com/en–us/downloads/49f1072b–c0ec–4b1e–bdd7–4661ea07b5b3/default.aspx. Last visited on May 2014.

[14] A. Queralt, A. Artale, D. Calvanese, and E. Teniente, "OCL-Lite: A decidable (yet expressive) fragment of OCL*," Proceedings of the 25th International Workshop on Description Logics (DL'12), Description Logics, vol. 846, 2012, pp. 312-322.

[15] B. Beckert, U. Keller, and P. H. Schmitt,"Translating the object constraint language into first-order predicate logic," Proceedings of the Workshop at Federated Logic Conferences (FLoC02), 2002, available at i12www.ira.uka.de/ key/doc/2002/BeckertKellerSchmitt02.ps.gz.

[16] J. Jaffar, M. J. Maher, K. Marriott, and P. J. Stuckey, "The semantics of constraint logic programs," J. Log. Program., vol. 37, 1998, pp, 1-46.

[17] A. Bundy, "Tutorial notes: reasoning about logic programs," Proceedings of the 2nd International Logic Programming Summer School (LPSS'92), LNCS, vol. 636, 1992, pp. 252-277.

[18] T. Baar, "The definition of transitive closure with OCL - limitations and applications," Proceedings of the 5th Andrei Ershov International Conference in Perspectives of System Informatics (PSI'03), LNCS, vol. 2890, 2003, pp. 358-365.

[19] B. Pérez, "Towards decision facts management systems: the particular case of clinical guidelines," PhD thesis, Department of Computer Science and Systems Engineering, University of Zaragoza, Spain, 2011.

[20] B. Pérez and I. Porres, "Authoring and verification of clinical guidelines: a model driven approach", Journal of Biomedical Informatics, vol. 43, num. 4, 2010, pp. 520-536.

[21] J. Oldevik,"MOFScript eclipse plug-in: metamodel-based code generation," Proceedings of the Eclipse Technology eXchange workshop (eTX) at the ECOOP 2006 Conference, Nantes, France, 2006.

[22] MOFScript user guide, version 0.6 (MOFScript v 1.1.11), 2006, Available at: http://www.modelbased.net/mofscript/docs/MOFScript-User-Guide.pdf. Last visited on May 2014.

[23] OMG, OMG document ad/2005-11-03. MOFScript second revised submission to the MOF model to text transformation RFP (2005), Available at: http://www.omg.org/. Last visited on May 2014.

[24] EMF development team, The eclipse modeling framework website: http://www.eclipse.org/modeling/emf/. Last visited on May 2014.

[25] The Eclipse UML2 project, Website: http://www.eclipse.org/modeling/mdt/?project=uml2. Last visited on May 2014.

[26] K. Anastasakis, B. Bordbar, G. Georg, and I. Ray, "UML2Alloy: a challenging model transformation," Proceedings of the 10th International Conference on Model Driven Engineering Languages and Systems (MoDELS'07), LNCS, vol. 4735, 2007, pp. 436-450.