Reasoning About UML/OCL Models Using 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*—The widespread adoption of Model Driven Engineering approaches has made of models to be cornerstone components in the software development process. This fact requires verifying such models' correctness 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 propose an overall framework to reason about UML/OCL models based on Constraint Logic programming (CLP). We use Formula as model finding and design space exploration tool. We show how to translate a UML model into a CLP program following a Meta–Object Facility (MOF) like framework. We enhance our proposal by identifying an expressive fragment of OCL, which guarantees finite satisfiability and we show its translation to Formula. We complete our approach by providing a Model Driven Architecture (MDA) based implementation of the UML to Formula translation. Our proposal can be used for software model design reasoning by verifying correctness properties and generating model instances of the modeled designs, using Formula.**

Keywords*—*UML; OCL; CLP; reasoning; verification

I. Introduction

The widespread adoption of Model Driven Engineering (MDE) approaches has made of models to be cornerstone components in the software development process. This fact requires verifying both the completeness and correctness of such models to ensure the quality of the final product, reducing time to market and decreasing development costs. In this context, UML and OCL constitute two of the most commonly used modeling languages. The Unified Modeling Language (UML) [11] has been widely accepted as the de–facto standard for building object-oriented software. The Object Constraint Language (OCL) [10], 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 flaws are not detected until the later implementation stages, thus increasing the cost of the development process [4]. 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 semantic–preserving translation to another language [4]. The resulted translation can be used to reason about the software design by checking predefined correctness properties about the original model [4].

In this paper, we advocate for using 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 [6], 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 [7]. 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 [9] to automatically generate the Formula specification from a CD. 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 which 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 use a model-to-text transformation tool to automatically transform a CD to Formula. Our framework can be used for software model design reasoning by checking correctness properties and generating model instances automatically using Formula, thus contributing to software designs' validation and verification.

The paper is structured as follows. Section II gives a brief introduction to Formula. An overview of our framework is presented in Section III. Section IV presents the translation of a CD to Formula, while Section V describes the chosen OCL fragment and its representation into Formula. The automatic MDA–based translation of a CD to Formula is presented in Section VI. Section VII summarizes the strengths and weaknesses of our approach and discusses related work. Finally, Section VIII covers our main conclusions and future work.

II. A Brief Overview of Formula

Formula distinguishes three units for modeling the problem: *domains*, *models* and *partial models*. A Formula *domain*

 ISBN: 978-1-61208-304-9

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 [6], 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 [6]. Formula allows to specify individual concrete instances of the design-space or parts thereof, in a specific Formula unit called *partial model* [6]. 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 [6].

```
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.

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 [6]. 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 an example of a *term*, in line 7 of Figure 1 we show `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 are filled with a do not-care symbol ('_'), so that Formula will find valid assignments. Terms are the basis for defining *predicates*, which constitute the basic units of data, used for defining *queries* and *rules*. An example of a predicate is `a1 is Association(name1,_,_,_,_,_,_)` (see line 7), where the variable `a1` is bound to the `Association` type.

Rules. Rules 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 [7]. 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 presented in the rule.

Queries. Corresponding to rules where left hand side is a nullary construction [7]. 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 [7]. 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, to ensure that upper bounds of associations' multiplicities are >= than lower bounds, we firstly need to define a query representing the existence of associations verifying the opposite (see line 6 of Figure 1). With this query, we are considering such incoherent situation as a valid state. Thus, to verify that such situation is invalid, we include the negation ('!') of the query in the `conforms` query (line 9).

Finally, 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 [6][7].

III. Encoding UML/OCL Models into Formula

As described previously, our proposal follows a MOF-like metamodeling approach, based on the framework the developers of the Formula tool give in [7]. Their framework provides a representation in Formula of part of the key concepts defined both at the MOF meta–level [11], representing the M2 level, and at the instance–level [11], 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 note that the authors in [7] 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

 ISBN: 978-1-61208-304-9

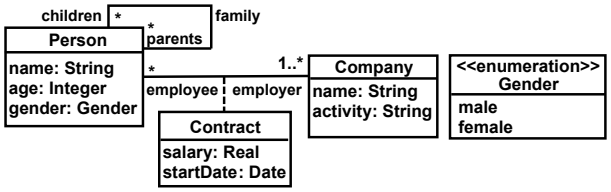

Figure. 2: Case study.

model instances of specific CDs, and not arbitrary ones as authors in [7] do, which we consider not enough when needed to reason about specific CDs. Thirdly, in contrast to [7], 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.

Each Formula unit defined in our approach contains two blocks of Formula expressions, related to the translation of the CD structural aspects (see Section IV) and its OCL constraints (see Section V), respectively. Our approach is illustrated with the case study of Figure 2, designed for explanation purposes covering basic aspects. It describes both the contractual relationship between a "Company" and a "Person", and the family recursive relationship connecting the class "Person".

IV. Translation of a CD's Structural Elements

This section presents a brief introduction of the rules we have defined to transform a class diagram ($\mathcal{CD}$), conforming with the UML metamodel [11] ($\mathcal{M}$), into Formula. Due to space reasons, in this paper we focus on a set of basic structural UML CD features (UML *class*, *attribute*, *association*) for being frequently used for modeling structural aspects of systems. Next, we briefly explain their translation classifying the generated Formula instructions into the different MOF levels. For the explanation, we lean on Table I.

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 [7], 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 [7], 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 [11], 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* [11]. 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* [11], so in this situations we name the created data types with the `Link` prefix. Since links connect class instances [11], 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 [11]. 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).

Finally, the Formula expressions resulted from the translation of a $\mathcal{CD}$ are grouped in 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.

 ISBN: 978-1-61208-304-9

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:* **primitive 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)**

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's 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 [4], [13]; (1) those which allow only specific kinds of constraints, (2) those which consider restricted models, (3) methods which do not support automatic reasoning, or (4) those which 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. Next, we introduce the chosen OCL fragment and give the main idea of its translation to Formula. Due to space reasons, we translate a simple OCL constraint, which will serve as a reference explanation for the remainder elements of our OCL fragment.

Introduction to the chosen OCL fragment. We consider the OCL invariant `context C inv: expr(self)`, where `C` is the class ϵ $\mathcal{CD}$ to which the invariant is applied and `expr(self)` is an OCL expression resulting in a `Boolean` value for each `self` ϵ `C`. 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 2), attributes' names (such as `c.name`, being `c` a `Company` instance in Figure 2), or operations (for example, `c.hireEmployee(p)`). Taking this into account, in Figure 3 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 `PathItem`, by combining roles' names, attributes' names or operations, with the dot operator. For an explanation of OCL, we refer to [10].

```
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. 3: Syntax of the OCL fragment.

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 working on a company 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 (FOL).

Second–step. Each constraint logic program P can be translated into FOL according to its *Clark Completion* P^* [8].

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:	$\neg(\exists$ `ageSlot(self,def,val) val<18).`(2)
Third–step:	`query:=ageSlot(self,_,val), val<18.` `conforms := ! query.` (3)

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. 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 [3], which compile specifications into the logic program it directly specifies. 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 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 `ageSlot` 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 ‘,’.

Thus, the translation of an invariant is carried out by means of the composition of the three defined functions. Next, we make some remarks regarding the translation of the remainder elements in our OCL fragment (see Table III). In particular, excluding the `select` and `transitive closure` elements, whose translation requires extra attention, we consider that the translation of the remainder OCL elements can be easily understood by considering our previous explanations.

Select operation. Since this operation refers to obtaining a subcollection from a set of elements, 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 rule: `FemaleEmp(self,sele) :- LinkContract(_,_,sele,self),genderPSlot(sele,_,val), val=female.`, which gathers only female employees.

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))`))

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 [2]. Nevertheless, it is possible to define the transitive closure of relations which are known to be finite and acyclic. In particular, for its translation we have based on both, the definition of transitive closure provided in [2], and the representation in CLP of acyclicity constraints provided in [7] (page 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 CD 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 CD into the Formula language, a professional with both UML and Formula skills may be required. Also, such an encoding process may entail a big effort depending on the CD 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 CD into the input language of a model finder tool a cumbersome and costly endeavor. To overcome these challenges, we use an MDA tool-approach, which allows us to automatically carry out the transformation from the CD to Formula. Among the large amount of MDA-based tools in the literature, we are interested in those with support for customizable model–to–text transformations. The idea is to define only one set of transformations for all CDs. Finally, we have chosen the MOFScript Eclipse plug-in [5], which we have already used in previous works [12]. In our

 ISBN: 978-1-61208-304-9

particular case, we use the UML 2.0 metamodel and the specific CD as the model, defined using the UML2 Eclipse plug-in [5]. As far as the Formula program generation is concerned, we have defined several MOFScript transformation scripts that generate the different Formula units with the translation of the structural CD elements. In particular, we have defined a set of MOFScript transformation rules, grouped into different MOFScript files, employed to produce the print Formula structures that constitute the three Formula units in our approach, which depend on the specific CD.

VII. Discussion and Related Work

As described previously, the formalization and analysis of UML CDs 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 [7] a comparison with other tools, both SAT (Boolean Satisfiability) solvers and alternatives such as *ECLiPSE* and *UMLtoCSP*, focusing mainly on Alloy [1], for being the closest tool to Formula. Although the Formula authors provide a careful comparison with Alloy in [7], 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 [7]. On the one hand, we propose a more faithful representation of the basic UML metamodel and instance domain elements [11]. We consider that providing a translation which 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 [7] 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 [13]. In [13] 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 CD 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`, `isEmpty/notEmpty`, `xor`, or `reject`). Finally, there are other operations (such as `oclIsTypeOf`, considered in [13]) that can not be represented into Formula, but 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. Our proposal can be used for model design reasoning by verifying correctness properties and generating model instances automatically using Formula. We provide an implementation of our CD to Formula proposal, being the implementation of the OCL fragment 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] K. Anastasakis and B. Bordbar and G. Georg and I. Ray, "UML2Alloy: A Challenging Model Transformation," Proc. of the 10th International Conference on Model Driven Engineering Languages and Systems (MoDELS'07), LNCS, vol. 4735, 2007, pp. 436-450.

[2] T. Baar, "The Definition of Transitive Closure with OCL - Limitations and Applications," Proc. of the 5th Andrei Ershov International Conference in Perspectives of System Informatics (PSI'03), LNCS, vol. 2890, 2003, pp. 358-365.

[3] A. Bundy, "Tutorial Notes: Reasoning about Logic Programs," Proc. of the 2nd International Logic Programming Summer School (LPSS'92), LNCS, vol. 636, 1992, pp. 252-277.

[4] J. Cabot and R. Clarisó and D. Riera, "Verification of UML/OCL Class Diagrams using Constraint Programming," Proc. of the 2008 IEEE International Conference on Software Testing Verification and Validation Workshop (ICSTW'08), IEEE Computer Society, 2008, pp. 73–80.

[5] Eclipse Modeling Project, Available at: http://www.eclipse.org/modeling/. Last visited on August 2013.

[6] Formula - Modeling Foundations, Website: http://research.microsoft.com/en-us/projects/formula. Last visited on August 2013.

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

[8] J. Jaffar and M. J. Maher and K. Marriott and P. J. Stuckey, "The Semantics of Constraint Logic Programs," J. Log. Program., vol. 37, 1998, pp, 1-46.

[9] OMG, OMG Model Driven Architecture, Document omg/2003-06-01, 2003, Available at: http://www.omg.org/. Last visited on August 2013.

[10] OMG, Object Constraint Language OCL, OMG Specification, Version 2.2, 2010, OMG Document Number: formal/2010-02-01. Available at: http://www.omg.org/spec/OCL/2.2. Last visited on August 2013.

[11] OMG, UML 2.4.1 Superstructure Specification, Document formal/2011-08-06. Available at: http://www.omg.org/. Last visited on August 2013.

[12] 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.

[13] A. Queralt and A. Artale and D. Calvanese and E. Teniente, "OCL-Lite: A Decidable (Yet Expressive) Fragment of OCL*," Proc. of the 25th International Workshop on Description Logics (DL'12), Description Logics, vol. 846, 2012, pp. 312-322.

 ISBN: 978-1-61208-304-9