Reasoning about UML/OCL Class Diagrams using Constraint Logic Programming and Formula

Beatriz Pérez[a,1], Ivan Porres[b]

[a]*Department of Mathematics and Computer Science, University of La Rioja, E-26004 – La Rioja, Spain*
[b]*Department of Information Technologies, Åbo Akademi University, FIN-20520 – Turku, Finland*

---

*Corresponding author
Email address: beatriz.perez@unirioja.es (Beatriz Pérez)

Summary

Model Driven Engineering promotes the use of models as the main artifacts in software and system development. Verification and validation of models are key activities to ensure the quality of the system under development. This paper presents a framework to reason about the satisfiability of class models described using the Unified Modeling Language (UML). The proposed framework allows us to identify possible design flaws as early as possible in the software development cycle. More specifically, we focus on UML Class Diagrams annotated with Object Constraint Language (OCL) invariants, which are considered to be the main artifacts in Object-Oriented analysis and design for representing the static structure of a system. We use the *Constraint Logic programming* (CLP) paradigm to reason about UML Class Diagrams modeling foundations. In particular, we use *Formula* as a model–finding and design space exploration tool. We also present an experimental Eclipse plug–in, which implements our UML model to Formula translation proposal following a Model Driven Architecture (MDA) approach. The proposed framework can be used to reason, validate, and verify UML Class Diagram software designs by checking correctness properties and generating model instances using the model exploration tool *Formula*.

Key words: UML ; OCL ; Constraint Logic Programming ; reasoning

1. Introduction

Model Driven Engineering (MDE) [1] promotes models as cornerstone components in software development. Verification and validation of models become important activities to ensure the quality of a system. Effective model verification and validation methods can reduce time to market and decrease development costs. In the context of MDE, the Unified Modeling Language (UML) and the Object Constraint Language (OCL) constitute two of the most commonly used modeling languages. UML [2] has been widely accepted as the de-facto standard object-oriented software modeling language. OCL [3] is an integral part of UML which has been introduced into UML as a declarative language to express integrity constraints that UML diagrams cannot convey by themselves.

Software models, as any other software artifact, may contain defects. Unfortunately, in some occasions, possible design flaws are not detected until the later implementation stages, thus increasing the cost of development [4, 5]. This situation requires a wide adoption of formal methods as well as verification and validation approaches. In this line, there have been remarkable efforts to formalize UML semantics, and to address and solve ambiguity, uncertainty, and underspecification issues detected in UML semantics. In particular, the formalization and analysis of specific UML artifacts can be done by carrying out a translation to another language that preserves the semantics [4, 5, 6, 7, 8, 9]. The resulting translation can be used for several purposes, such as to reason about implicit properties in UML models and about particular model instances.

In this paper we propose an overall framework to reason about specific UML Class Diagram (CDs) based on the *Constraint Logic programming* (CLP) paradigm. More specifically, we focus on UML Class Diagrams, annotated with OCL constraints, which are considered to be the mainstay of object-oriented analysis and design representing the static structure of a system, and whose formalization and analysis have motivated a significant number of proposals [4, 10, 11]. As reasoning tool, we use a model–finding and design space exploration tool called *Formula* [12], which presents distinctive strength properties compared to other similar tools, including more expressivity [13, 14]. More specifically, Formula is based on algebraic data types and CLP, and relies on the *Formula solver Z3* as underlying engine to reason about models where proof goals are encoded as CLP satisfiability problem. Formula utilizes a bounded verification approach by means of which the reasoning process is carried out by establishing finite bounds for the number of instances of the model to be considered during the verification process. In the case that *Z3* finds a solution that satisfies all encoded constraints, Formula will reconstruct a complete model from this information derived of known facts.

Our approach can be used for several different purposes. It can be used to rigorously reason about a UML design, by checking predefined correctness properties about the original model, such as satisfiability or the lack of redundant constraints [5]. Additionally, our proposal can be used to inspect models of complex system development contexts, to search for conforming object models and to choose those that better fit domain needs. Overall, our proposal can contribute to software design validation and verification.

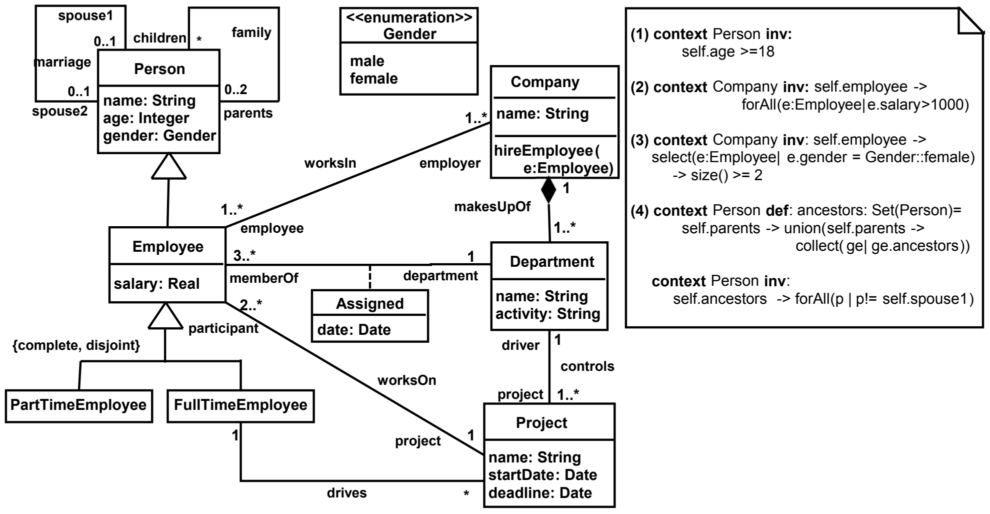
The results presented in this paper are based on the work published by the authors of this paper in [15, 16]. In this paper we provide a revised and extended version of those works, focusing mainly on the conceptual definition of our framework, constituting the first paper that includes a complete description of our proposal. Additionally, in this paper we provide extra material regarding the reasoning process to follow when using our approach and a more detailed explanation about the comparison among our proposal and others.

The paper is structured as follows. Next, we motivate and present an overview of our approach, introducing the case study we use throughout the paper. Section 3 provides a brief introduction to Formula. Section 4 presents the translation of a UML class diagram to Formula, while Section 5 describes the OCL fragment we consider in our proposal and its representation into Formula. Section 6 describes the *CD2Formula* tool we have developed to implement our Class diagram to Formula translation proposal, and illustrates the usefulness of our overall approach by applying it to the case study. Section 7 summarizes the strengths and weaknesses of our approach and discusses related work. Finally, Section 8 contains our main conclusions.

2. Motivation and overview

2.1. The need for class diagram verification and validation

In order to motivate our proposal, we build upon the case study shown in Figure 1 to identify inconsistent modeling features. This class diagram has been designed for explanation purposes and covers a representative number of UML Class Diagrams elements from those our approach supports. More specifically,



```
(1) context Person inv:
        self.age >=18

(2) context Company inv: self.employee ->
                    forAll(e:Employee|e.salary>1000)

(3) context Company inv: self.employee ->
    select(e:Employee|  e.gender = Gender::female)
        -> size() >= 2

(4) context Person def: ancestors: Set(Person)=
        self.parents -> union(self.parents ->
                              collect( ge| ge.ancestors))

    context Person inv:
        self.ancestors  -> forAll(p | p!= self.spouse1)
```

Figure 1: Case study.

it describes part of the organizational and functional structure of companies regarding their employees, departments, and the projects undertaken by the companies. The whole/part strong dependency between a company and its departments has been represented by a strong composition. The relationship between an employee and the department to which he/she is assigned is represented by the association class "Assigned". The CD also registers the projects controlled by each department and the employees who work on each project, including the employee driver of each project. In order to make the example more interesting, a three-level hierarchy has been considered to represent different types of people in the system. The "family" and "marriage" relationships among people registered in the system have also been represented. A set of business rules has been established by OCL constraints, which particularly will be used to explain our proposal for the translation of Class Diagram constraints.

In particular, as we will see later, our proposal could help us to detect unsatisfiable models. As an example, let's consider the following constraint in the case study of Figure 1: "every department has a project with an only employee as participant". Such a constraint can be defined in OCL as:

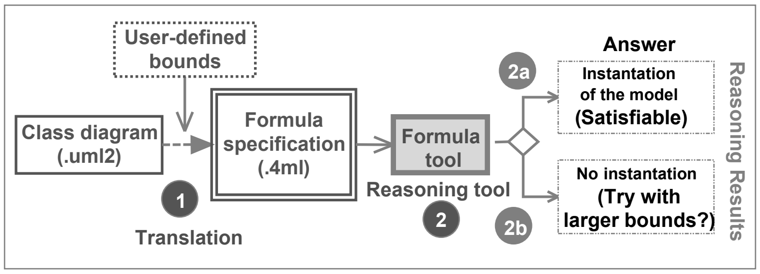

User-defined bounds
Class diagram (.uml2)
Formula specification (.4ml)
Formula tool
Reasoning tool
1
Translation
2
2a
2b
Answer
Instantation of the model (Satisfiable)
No instantation (Try with larger bounds?)
Reasoning Results

Figure 2: Architecture of the framework.

```
context Department inv:  self.project->
             exists(p:  Project | p.participant -> size()=1)
```

Considering this constraint, the overall model would be unsatisfiable due to the conflict of this constraint, which forces each department to have a project with a single participant, and the multiplicity of "Employee" in the association "worksOn", which forces each project to have at least two participants (`2..*`). Frameworks like the one we propose could help us to detect these situations, which motivates their use.

2.2. Proposed solution

The overall framework we propose to reason about class diagram models annotated with OCL constraints consists of two steps: (1) translating the class diagram model to the Formula language and, (2) using Formula for reasoning about such a model. The overview of our framework is represented diagrammatically in Figure 2.

2.2.1. First step. From the Class diagram model to the Formula language

First, we need to translate the class diagram, annotated with OCL constraints, which we want to reason about, into the input specification language of the Formula tool (see step number 1 in Figure 2). On the one hand, the translation of the class diagram to Formula is carried out by following the guidelines explained in Section 4. In this step, the user would have to manually indicate the number of valid instances (user–defined bounds) of the class diagram the user desires Formula to generate as part of the resulting instantiation of the model (that is, the object diagram).

On the other hand, the translation of CD constraints to Formula is performed as described in Section 5. Since OCL constraints are essentially first order predicate logic statements [6], and validity in FOL is undecidable (also known as Church's Theorem, see the survey paper [17]), checking the correctness of OCL constraints is an undecidable problem [5, 6]. Therefore, we have identified a fragment of OCL, which can be checked for finite satisfiability, while being considerably expressive. In Section 5 we also show how to translate such an OCL fragment to Formula by giving, as an intermediate step, a representation of the OCL constraints as First-Order Logic (FOL) expressions.

More specifically, our class diagram to Formula translation proposal follows a MOF-like metamodeling approach [2], based mainly on the proposal the developers of the Formula tool gave in [14, 18]. Their proposal provided a representation in Formula of part of the key concepts defined both at the M2 meta-level, and at the M1 model-level [2]. The resulting Formula expressions are grouped in a Formula unit, which is used by the *Formula solver Z3* to find, if it exists, a valid set of instances of arbitrary CDs at the M1 level (conforming with their M2 representation) and its corresponding instances representing the OD at the M0 level (conforming with their M1 representation). We note that the authors in [14, 18] did not provide a representation approach of specific OCL constraints included in the UML model.

Based on this proposal, we have extended and modified it, giving weight to three main aspects. First, we have mainly focused on obtaining a more faithful representation of the level–based distribution, specifying a richer metamodeling framework. Our extended proposal is materialized into four different Formula units distributed along the M2, M1 and M0 levels, which ease the application and understandability of our approach, while promoting unit reutilization. We also give support for the translation of more UML model elements (such as user-defined data types, including enumeration types, multiplicities of properties, strong composition or full support of generalization). Second, in contrast to [14, 18], we have developed an approach for the translation of OCL constraints to Formula, which (1) identifies a significantly expressive fragment of OCL, and (2) provides a translation into Formula of OCL constraints defined by such a fragment (or OCL equivalent expressions). Finally, in order to provide tool support of our proposal, we have developed an Eclipse plug-in called *CD2Formula* based on MDA, which implements our CD to Formula translation approach to easily and automatically perform the translation process.

2.2.2. Second step. Reasoning process

Once the CD model has been translated into the Formula language, the Formula finder is used to detect whether the model is satisfiable (see step number 2 in Figure 2). At this point, if positive, the tool returns an instantiation of the model, verifying all the established constraints (see substep 2a). Otherwise, the Formula tool does not return an instantiation of the model which, however, does not constitute a proof of unsatisfiability beyond the analyzed domain, that is, it does not necessarily mean that the model is not satisfiable in general (see substep 2b). More specifically, in this latter situation when no instantiation model is returned, the tool shows an "unsatisfiable" label together with a mark on the Formula queries that are not satisfied by the model considering the given bounds. Such queries can give the user a clue about whether the result has been motivated (1) by a problem in the model definition, because it would be indeed unsatisfiable, or (2) by the chosen bounds, which could motivate the user to retry the process changing the bounds in order to make subsequent analysis.

Overall, as described previously, our framework can be used both for software model design reasoning by checking correctness properties and for generating model instances automatically using Formula, thus contributing to software design validation and verification.

3. A brief overview of Formula

Formula distinguishes three different units to represent a system: *domains*, *models*, and *partial models*. Modeling in Formula always starts with specifying

```
1 domain MetaLevel extends UserDataTypes {
2 [Unique(name ->isAbstract)]
3 primitive Class ::= (name: String, isAbstract: Boolean).
4 [Closed(srcType, dstType)]
5 primitive Association ::= (name:String,
    srcType:Class, srcLower:Natural, srcUpper:UpperBound,
    dstType:Class, dstLower:Natural, dstUpper:UpperBound).
6 Classifier ::= Class + Association.
7 error_meta_BadMultInterval := Association(_,_, srcL,srcU,_,_,_),
                                             srcL >srcU.
8 error_meta_dupAssoc := a1 is Association(name1,_,_,_,_,_,_),
                          a2 is Association( name2,_,_,_,_,_,_),
                                          name1 = name2, a1 != a2.
9 ...
10 conforms:= !error_meta_BadMultInterval &
                                          ! error_meta_dupAssoc & ...}.
```

Figure 3: An extract of a Formula domain.

the *problem domain* and formalizing an abstraction of the problem that can be used by Formula to reason about the design [12]. A Formula *domain* (FD) is the basic specification unit for an abstraction and allows specifying algebraic data types and a logic program describing properties of the abstraction. As an example, line 3 in Figure 3 shows the definition of an FD called `MetaLevel` containing an algebraic data type named `Class`. The CLP paradigm provides a formal and declarative approach for specifying such abstractions [12], which in Formula are represented by *rules* and *queries* (which we will explain later). Domains can extend other domains by including the `extends` keyword (see line 1 in Figure 3 where the `MetaLevel` domain extends the domain `UserDataTypes`).

A Formula *model* (FM) is a finite set of data type instances built from constructors defined in the associated domain FD and satisfies all FD constraints [18]. As an example, the Formula expression `Class("Person", false)` would correspond to an instance of the data type `Class` described previously. Formula allows specifying individual concrete instances of the design-space or parts thereof, in a specific Formula unit called *partial model*.

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 FM that must be solved [12]. Partial models allow unknowns to be combined with parts of the model that are already fixed [18]. They are essentially lower bounds on the type of models we want to find. Fixed parts of a model can be included in the partial model explicitly, specifying the corresponding Formula instructions inside the model, or implicitly by an "including" instruction (using the `includes` keyword) in the head of the partial model. Additionally, it is necessary to specify the domain(s) the partial model conforms by using the `of` keyword. Partial models can include different generation options that Formula provides for search configuration, and which are based on the use of search space boundaries. An example of such generation options is the use of the `Introduce(f,n)` option, which adds at most `n` terms of the form `f` to the partial model. For example, the instruction `Introduce(Class,2)` would cause Formula to generate at most two arbitrary instances of the `Class` element. The values allowed for `n` are positive integers or zero. If improper values are set for such `n` terms (such as zero or negative integers), the Formula tool detects it as an error when loading the model into the tool interpreter.

An FD consists of *algebraic data types*, *rules*, and *queries*. First, *algebraic 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 [12]. In general, *rules* specify implications and *queries* restrict the valid states by specifying forbidden states. Next, we explain the main characteristics of these Formula constructors.

Algebraic data types. They are defined by the operator *::=*, indicating on the right hand side their properties by *fields*. Properties in data types are defined by means of *fields*, which must be of some concrete type (Formula built–in types or other user data types). Data types can be labeled in their definition with the `primitive` keyword, defining *primitive constructors*, which intuitively can be used to extend the program taking part in other type definitions. Otherwise, the data type definition results in a *derived constructor*. As an example, the definition of the primitive data type `Class` is illustrated on line 3 in Figure 3. This data type defines several *fields* together with their types (such as field `name` of type `String`). Furthermore, the derived type `Classifier` is defined as the union (+) of the `Class` and `Association` types (see line 6 of Figure 3).

Additionally, constants are defined using the operator ::=. This operator can be used to define data types with a fixed value or a list of fixed values within curly brackets. For example, the constant `Star` defined as `Star::={star}`, would represent the unspecified upper bound in the multiplicities of associations.

Around data types, Formula defines different categorizations of structural elements as building blocks for defining Formula expressions. These elements are mainly *terms* and *predicates*. As an example of a *term*, on line 8 of Figure 3 we list `Association(name1,_,_,_,_,_,_)`, which represents all instances of the `Association` term, where the first field is set to a fixed property (`name1`). The other fields are filled with a do-not-care symbol ('_'), so that Formula will find valid assignments. Terms are the basis for defining *predicates*, which constitute basic units of data, used for defining *queries* and *rules*. An example of a predicate is `a1 is Association(name1,_,_,_,_,_,_)` (see line 8), where the variable `a1` is bound to the `Association` type.

Additionally, Formula allows using different *annotations* in the definition of data types to *reduce the size of the search space*. For example, the `[Closed]` annotation, whose syntax is `[Closed(DT fields)]`, which instructs Formula to apply a closed check to instances of the corresponding data type (`DT`) that is, using only the instances of that type given in the model. Otherwise Formula would be able to invent new instances, which is a desired behavior for general model-finding problems. An example of the `[Closed]` annotation is illustrated on line 4 in Figure 3 where it ensures that Formula instances of associations are created by class instances that exist in the model. Additionally, `[Unique]`, whose syntax is `[Unique(DT fields -> DT fields)]`, requires all records with identical fields on the left of the arrow (`->`) to have identical fields to the right of the arrow. By adding the `[Unique]` attribute to a constructor type definition, Formula introduces new queries to the containing domain, which ensure that an element of the domain of the relation is mapped to a single element of the codomain. As an example, the `[Unique]` annotation on line 2 in Figure 3 checks that there are not two class instances with the same field values.

Rules. A *rule* behaves like a universally quantified implication; whenever the relations on the right hand side of a rule hold for some substitution of the variables, then the left hand side holds for that same substitution [14, 18]. The intuition behind rules is *production*; they create new entries in the fact-base of Formula, populating previously defined types with facts representing the members in the collection presented in the rule. Rules are specified by means of the operator `:-`, indicating in the left–hand of the expression a simple term and on the right–hand the list of *predicates* specifying the rule.

Queries. A *query* corresponds to a rule where the left–hand side is a nullary construction [14, 18]. 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 [14, 18]. Queries are constructed using the operator `:=`, joining Formula *predicates* that specify the forbidden states. Additionally, queries can be used like propositional variables to construct other queries. In particular, Formula defines in every domain the `conforms` standard query, where all constraints come together and define a valid instance of the domain. 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 multiplicities in associations are greater than or equal to lower bounds, we first need to define a query representing the existence of associations verifying the opposite (see query `error_meta_BadMultInterval` on line 7 of Figure 3). With this query, we consider such an incoherent situation as a valid state. Thus, to verify that such a situation is not valid, we need to include the negation ('!') of the query in the `conforms` query (see line 10 of Figure 3).

Finally, to explore the design–space, Formula loads the domains and the partial models defined for the specific problem and executes the logic program. The execution locates all intermediate facts that can be derived from the given facts in the partial models, and attempts to find valid assignments for the unknowns. Formula relies on the *Formula solver Z3* to carry out this step. In the case that *Z3* finds a solution that satisfies all encoded constraints, Formula will reconstruct a complete model from this information derived of known facts [12].

4. Translation of Class Diagram structural elements

This section presents a brief introduction of the rules we have defined to transform a CD, conforming with the UML metamodel [2], into Formula. First, we explain the translation of a set of basic structural UML Class Diagram features frequently used for modeling structural aspects of systems (UML *class*, *property*, bidirectional *association*, and *generalization*, considering also the different types of generalization set constraints). We finish the section with the translation of other elements, such as *classifiers*, *association classes*, *strong composition*, and *user–defined* data types (including *enumeration* data types). Thus, in this paper, we provide a more detailed and complete explanation of the proposal we presented in [15, 16] where the elements *class*, *property*, bidirectional *association*, *classifier* and *association class* were briefly handled.

The Formula instructions generated from the translation of the CD elements are classified into the M2, M1 and M0 levels. For the entire explanation, we rely on Table 1 (to explain the elements in the M2 level), in Tables 2 and 3 (to present the elements in the M1 level), and in Table 5 (to explain the elements in the M0 level). In these tables, we represent in bold font the fixed elements in the translation. To allow the reader a better understanding of our approach, we reinforce our explanation for the specific translation of *classes*, and in particular the class *Person* of our case study, illustrating it in Figure 4. In this figure, the four Formula units defined in our approach are represented by rectangles, which include the transformation patterns defined in each case, while the Formula expressions resulted from the application of such patterns are depicted by rhomboids. To improve readability, we also represent the arity of each Formula data type as *dataTypeName/n*, where *n* is the arity of the data type *dataTypeName*.

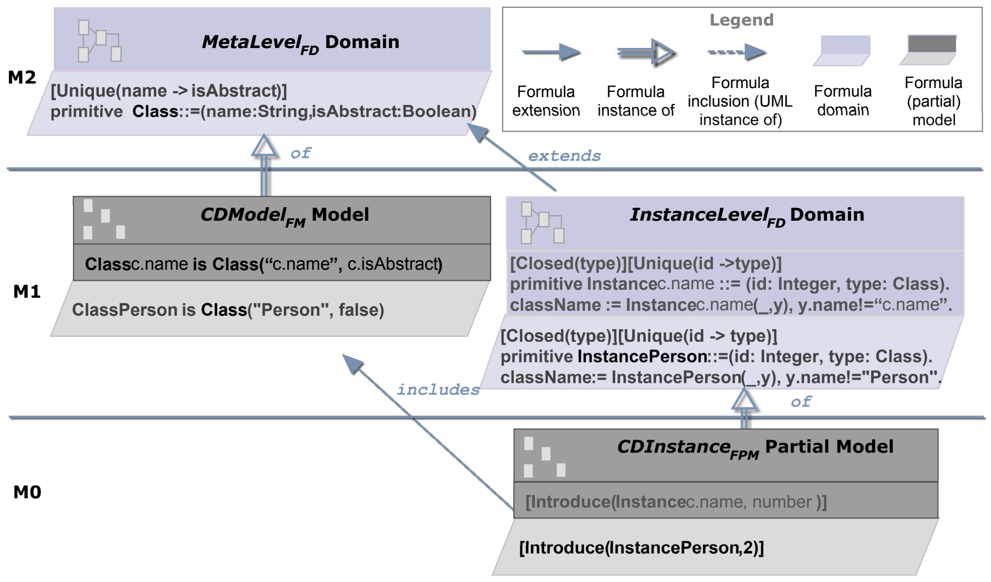

Legend
Formula extension
Formula instance of
Formula inclusion (UML instance of)
Formula domain
Formula (partial) model
M2
MetaLevel_FD Domain
[Unique(name -> isAbstract)]
primitive Class::=(name:String,isAbstract:Boolean)
of
extends
M1
CDModel_FM Model
Classc.name is Class("c.name", c.isAbstract)
ClassPerson is Class("Person", false)
InstanceLevel_FD Domain
[Closed(type)][Unique(id ->type)]
primitive Instancec.name ::= (id: Integer, type: Class).
className := Instancec.name(_,y), y.name!="c.name".
[Closed(type)][Unique(id -> type)]
primitive InstancePerson::=(id: Integer, type: Class).
className:= InstancePerson(_,y), y.name!="Person".
includes
of
M0
CDInstance_FPM Partial Model
[Introduce(Instancec.name, number)]
[Introduce(InstancePerson,2)]

Figure 4: Formula expressions generated for classes.

4.1. Level M2.

For each metamodel element *Class*, *Association*, *Property* and *Generalization*, we define a primitive Formula data type with the same name and specific *fields* (see Table 1). For example, in the case of the *Class* metamodel element, we define the data type `Class/2` with two fields: `name`, of type `String`, and `isAbstract`, of type `Boolean` (see Table 1 as well as in Figure 4). In the case of the *Association* metamodel element, we define the data type `Association/7` with several fields representing the name, the associated classes (`srcType` and `dstType`), and the multiplicities of such classes in the association. Regarding the *Property* metamodel element, we define a type for each *built–in type*, called *`typeName`*`Property/4`, with specific fields (see Table 1). In addition to `Integer`, `String`, and `Boolean`, included in [18], we also give support to `Real`, `LiteralNull`, and `UnlimitedNatural` types. As stated in [2], we also consider `lower/upper` bounds representing property multiplicity constraints. Furthermore, a derived data type named `Property` is created as the union of all types of properties to be used as a generic property type. The data type `HasProperty/2` is

Table 1: Excerpt of the proposal regarding M2 level.

M2 level	
Class	**Association**
[Unique(name -> isAbstract)] primitive Class ::= (name: String, isAbstract: Boolean).	[Closed(srcType, dstType)] primitive Association ::= (name: String, srcType: Class, srcLower: Natural, srcUpper: UpperBound , dstType: Class, dstLower: Natural, dstUpper: UpperBound).
Property	**Generalization**
primitive StringProperty ::=(name:String, def:String, lower:Natural, upper: UpperBound). ... primitive LiteralNullProperty ::= (name: String, def: Null,...). primitive UnlimitedNaturalProperty ::= (name:String, def: UnlimitedNatural ,...) Property::= StringProperty + ...+userDataTypeProperties . [Closed(owner, prop)] primitive HasProperty ::= (owner: Classifier, prop: Property).	[Closed(sup, sub)] primitive Generalization ::= (sup: Class, sub: Class). supClass ::= (sup: Class, sub: Class). supClass (x, y) :- Generalization(x, y). supClass (x, z) :- supClass (x, y), supClass (y,z). inhsProp ::= (owner: Classifier, prop: Property). inhsProp (cl, prop) :- HasProperty (cl, prop). inhsProp (csub, prop) :- supClass (csup, csub), HasProperty (csup, prop). inhsAsso ::= (owner: Classifier, asso: Association). inhsAsso (cl, a) :- a is Association(_,cl,_,_,_,_,_), ... Generalization(cl, _). inhsAsso (csub, Association(a.name ,csub,a.srcLower ,...)) :- supClass (csup, csub), inhsAsso (csup, a), ... a is Association(_, csup,_,_,_,_,_).

also defined to represent the possession of a property by a classifier.

Finally, we represent the *Generalization* metamodel element by the data type `Generalization/2`, with two fields of the data type `Class` previously defined (`sup` and `sub`) representing the superclass and subclass of the generalization (see Table 1). It is worth noting that, in order to represent additional semantics of generalizations, we create specific Formula expressions for several purposes. First, to allow Formula to generate the complete structure of inheritance from direct relationships (if C specializes B, and B specializes A, then C specializes A), we define a new data type `supClass`, together with two rules which allow Formula to populate the `supClass` data type with facts representing the overall structure of inheritance (see translation for generalizations in Table 1). Similarly, in order to hold inherited properties and associations, we define two new data types `inhsProp` and `inhsAsso`, together with two–four rules, respectively, which allow Formula to populate such new data types with facts representing inherited properties and associations. These Formula rules allow us to give support for multiple inheritance, since such rules create new facts representing inherited properties and association facts from superclasses to subclasses (in contrast to authors in [14, 18], which do not consider associations' inheritance in subclasses, but just properties' inheritance). To sum up, these data types allow Formula to create Formula instances representing specific UML classes, associations, types of properties, and generalizations at the M1 level.

Furthermore, we have used Formula `[Unique]` and `[Closed]` annotations so that Formula reduces the size of the search space when finally exploring the design–space. In the case of the UML *Class* metamodel element, the Formula `[Unique]` attribute is applied, ensuring that there are not two identical instances of the `Class` type. For the UML *Association* element, the `[Closed]` attribute is applied to the `Association` data type to instruct Formula to use only the instances of that type, given in the model (see Table 1). A `[Closed]` annotation is also applied to the `HasProperty` and `Generalization` types. We note that these constraints refer to Formula restrictions, not to CDs' constraints.

Finally, the Formula expressions defined at this *metamodel level* (M2) are included in an FD called $MetaLevel_{FD}$. As an example, see the definition of the `Class` data type in level M2 of Figure 4, enclosed in the $MetaLevel_{FD}$ domain. Since the representation of the meta–level M2 is the same whatever CD is considered, this FD is defined once and used for each CD. An excerpt of the $MetaLevel_{FD}$ domain has been presented in Figure 3.

4.2. Level M1.

At this level we define two groups of expressions denoted by *M1a* and *M1b*, respectively. These expressions will be enclosed, as we will explain later and depict in Figure 4 for the particular case of *classes*, in the following Formula units: $CDModel_{FM}$ model and $InstanceLevel_{FD}$ domain, respectively.

[M1a.] Each specific *class*, *association*, *property*, and *generalization relationship* in the CD, is represented by a Formula instance of the corresponding constructor (`Class`, `Association`, `Property`, or `Generalization` defined at level M2). With these Formula instances, we are explicitly representing specific elements in a CD. For example, the elements `ClassPerson`, `family`, and `genPersonEmployee` defined in Table 2 correspond to three Formula instances of the constructors `Class`, `Association`, and `Generalization`, respectively, defined at M2. At this

Table 2: Excerpt of the proposal regarding M1 level, group M1a.

M1 level- Group M1a	
Class	**Association**
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, star, Class("Person",false), 0, 2)
Property	**Generalization**
Translation pattern: p.name+p.owner.name **is** p.type**Property("**p.name**",**p.default, p.lowerValue,p.upperValue**)** **HasProperty(**owner,p.name+p.owner.name**P)** *Example:* namePerson is StringProperty("name","",1,1) HasProperty(Class("Person",false), StringProperty("name","",1,1))	*Translation pattern:* **gen**g.general.name**+**g.specific.name **is** **Generalization(Class**g.general.name**,** **Class**g.specific.name**)** *Example:* genPersonEmployee is Generalization(ClassPerson, ClassEmployee)

point we want to note that each specific association is required to have a unique name so that the corresponding defined Formula instance could be uniquely identified. Specific *properties* in the CD are represented by a Formula instance of the corresponding `Property` constructor (e.g., *`namePerson is String`*`Property(‘‘name’’,‘‘’’,1,1)` in Table 2, where the pair `1,1` represents that a person can have one and only one name), and by an instance of the data type `HasProperty`, representing the property's ownership (see Table 2).

As advanced previously, the Formula expressions defined in *M1a* constitute the Formula model called $CDModel_{FM}$. This model conforms with the $MetaLevel_{FD}$ domain, defined at level M2 (see Figure 4), in the same way as the element `ClassPerson` defined in the left hand rhomboid in level M1 of Figure 4 constitutes a Formula instance of the constructor `Class`, defined at M2.

[M1b.] So that Formula is able to generate instances of the specific *classes*, *associations*, *properties*, and *generalization relationships* in the CD to explore the concrete design–space, we need to create specific Formula data types representing each type of instance. For the definition of these types, we have based on the description of the *Instances* package [2], in particular, on the *Instance-Specification* element for classes, associations, and generalization relationships, and on the *Slot* element for properties. On one hand, the definition of the UML

InstanceSpecification element includes the classifier of the represented instance and the associated *InstanceValue* [2]. Taking this into account, for each *class c* in the CD, we define a primitive Formula data type called `Instance`*c.name*`/2` , with two fields, representing the associated classifier (`type`), and representing the instance value (`id`), respectively (see Table 3). As an example, see the primitive data type `InstancePerson` in Table 3. When the classifier is an association, the UML *InstanceSpecification* element describes a *link* [2], so in these situations we name the created data types with the `Link` prefix. Since links connect class instances [2], for each *association a* in the CD, we define a primitive Formula data type called `Link`*a.name*`/4`, which also includes references to the associated classes (see for example `LinkFamily` in Table 3). For each *generalization* relationship parent-child *g* in the CD, we define a specific `Link` data type in order to distinguish generalizations from association relationships. In particular, we create a primitive Formula data type called `LinkGen`*g.general.name+g.specific.name*`/4`, which particularly includes references to the associated classes, in this case, the super and sub classes (see for example `LinkGenPersonEmployee` in Table 3). We note that the definition of this particular link data type is needed by Formula in order to represent hierarchies at the instance level. Finally, so that Formula can generate property's specific values, we define specific data types representing the property's slots, based on the UML specification of the *Slot* element [2]. Taking this into account, for each *property p* in the CD, we define a primitive type called *p.name+p.owner.name*`Slot/3`, which registers the owner, the property type, and its value (e.g., `namePersonSlot`).

Formula requires establishing specific constraints to the defined data types. For example, in the case of each `Instance`*c.name*`/2` data type, we define a specific query to ensure that the instances of the data type have, as `type` value, an instance of the corresponding *class c* in the CD. That is, instances of a person, such as *InstancePerson(1, ClassCompany)* are not admissible but, for example, instances of *InstancePerson(1, ClassPerson)* are admissible. This constraint is imposed by the definition of the `className` query (see Table 3) and the inclusion of its negation in the final `conforms` query. Similar queries are defined in the

Table 3: Excerpt of the proposal regarding M1 level, group M1b.

M1 level- Group M1b	
Class	**Association**
Translation pattern: **[Closed(type)][Unique(id ->type)]** **primitive Instance**c.name **::= (id: Integer, type: Class).** **className := Instance**c.name**(_ ,y), y.name!=**“c.name”**.** *Example:* [Closed(type)][Unique(id ->type)] primitive InstancePerson::=(id: Integer, type: Class). className:= InstancePerson(_,y), y.name!=“Person”.	*Translation pattern:* **[Closed(type**,a.memberEnd.at(1).name **->** a.memberEnd.at(2).name**)]** If it is not a many–to–many relationship **[Unique(**a.memberEnd.at(1).name **->** a.memberEnd.at(2).name**)]** **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**).** **associationName:= Link**a.name**(_,aso,_,_),aso.name!= “**a.name**”.** *Example:* [Closed(type, children -> parents)] primitive LinkFamily::=(id:Integer,type:Association, children:InstancePerson, parents:InstancePerson). associationName:= LinkFamily(_,aso,_,_), aso.name!=“family”.
Property	**Generalization**
Translation pattern: **[Closed(owner,prop)][Unique(owner, prop ->value)]** **primitive** p.name+p.owner.name**Slot ::= (owner:Element, prop:**p.type**Property, value**: valueType**)** **slotName :=**p.name+p.owner.name**Slot(_,prop,_), prop.name!=** “p.name”. **slotOwner :=**p.name+p.owner.name**Slot(owner,_,_), owner.type.name!= “**p.owner.name**”.** *Example:* [Closed(owner, prop)] [Unique(owner, prop->value)] primitive namePersonSlot:= (owner: Element, prop:StringProperty, value:String). slotName:= namePersonSlot(_,prop,_), prop.name!=“name”. slotOwner:= n is namePersonSlot(owner,_,_), owner.type.name!=“Person”.	*Translation pattern:* **[Closed(type,** g.specific.name, g.general.name**)]** **[Unique(**g.specific.name**->**g.general.name**)]** **[Unique(**g.general.name **->**g.specific.name**)]** **primitive LinkGen**g.general.name**+**g.specific.name **::= (id: Integer, type:Generalization,** g.general.name **: Instance**g.general.name g.specific.name **: Instance**g.specific.name**).** **genType:= LinkGen**g.general.name+g.specific.name**(_,gen,_,_), gen.sup.name!= "**g.general.name**".** **genType:= LinkGen**g.general.name+g.specific.name**(_,gen,_,_), gen.sup.name!="**g.specific.name**".** **error_GenOneAndOnlyOne:= c is Instance**g.specific.name**,** **count(LinkGen**g.general.name+g.specific.name**(_,_,p,c))!=1.** **If the generalization set constraint is*complete* and/or *disjoint* *include the corresponding queries*** *Example:* [Closed(type, Employee, Person)] [Unique(Employee->Person)][Unique(Person ->Employee)] primitive LinkGenPersonEmployee ::= (id: Integer, type:Generalization, Person:InstancePerson Employee:InstanceEmployee). genType:= LinkGenPersonEmployee(_,gen,_,_), gen.sup.name!= "Person". genType:= LinkGenPersonEmployee(_,gen,_,_), gen.sub.name!="Employee". error_GenOneAndOnlyOne:= c is InstanceEmployee, count(LinkGenPersonEmployee(_,_,p,c))!=1.

case of associations (`associationName`), properties (`slotName` and `slotOwner`), and generalizations (`genType`). Additionally, `[Closed]` and `[Unique]` constraints are included. More specifically, in the case of associations, the `[Unique]` constraint is imposed only in associations which are not *many to many*, since in this type of associations there can be more than one association instance between a pair of classes `a-b` (see translation pattern in Table 3). Regarding generalizations, we define a `[Closed]` constraint to reduce the size of the search space and two `[Unique]` to ensure that there are not two identical instances of the `LinkGen` type. Again, these constraints refer to Formula restrictions, not to CDs' constraints.

In the particular case of generalizations, we have to include a specific Formula constraint in order to make sure that, in each generalization relationship, each instance of the child is associated with one and only one instance of the parent (see the definition of the query `error_GenOneAndOnlyOne` in Table 3 representing the opposite semantics). Again, its negation is included in the `conforms` query for the verification of the original constraint.

A specific remark has to be made regarding generalization sets. As stated in [2], each *Generalization Set* defines a particular set of Generalization relationships that describe the way in which a general Classifier (or superclass) may be divided using specific subtypes. For example, in our case study a generalization set defines a partitioning of the class `Employee` into the two subclasses: `PartTimeEmployee` and `FullTimeEmployee`. `Employee` could also have been divided into `MaleEmployee` and `FemaleEmployee` which would define a different generalization set. In particular, UML defines four constraints that may be applied to generalization sets: *complete/incomplete* and *disjoint/overlapping* [2, 19]. Regarding *complete/incomplete* constraints:

◇ *complete* specifies that all children (subclasses) in the generalization set have been specified in the model and no additional children are permitted.

◇ *incomplete* represents the fact that not all children (subclasses) in the generalization set have been specified and additional subclasses are permitted.

As for as the *overlapping/disjoint* constraints is concerned:

◇ *overlapping* indicates that instances of the parent (superclass) in a generalization set may have more than one of the children (subclasses) as a type; that is, their intersection is not empty.

◇ *disjoint* represents the fact that instances of the parent (superclass) in a generalization set may have no more than one of the children (subclasses) as a type; that is, their intersection is empty.

Based on these different constraints, the four different types of generalization sets constraints are: {`complete, disjoint`}, {`incomplete, disjoint`} (which corresponds to the default option in UML), {`complete, overlapping`}, and {`incomplete, overlapping`}. We have represented these constraints in Formula, taking into account their specific semantics [2, 19]. Since the *complete* and *disjoint* partitions are more restrictive, we directly represent them in Formula through the definition of specific queries. More specifically, for each generalization set with the *complete* constraint, we define the Formula query `queryComplete` (see Table 4),

Table 4: Generalization sets constraints and example of use.

Translation of a complete partition in a generalization set (GS)
- Define a queryComplete query as:
queryComplete:= p is Instance*Parent*, fail LinkGen*ParentChild*$_1$(_,_,p,_),
fail LinkGen*ParentChild*$_2$(_,_,p,_), ...
fail LinkGen*ParentChild*$_n$(_,_,p,_).
- Include its negation (!) in the conforms query.
Translation of a disjoint partition in a generalization set (GS)
- Per each disjoint pair of subclasses (*Child*$_i$,*Child*$_j$) in each generalization set (GS) define:
queryDisjoint*GSName*$_k$:= LinkGen*ParentChild*$_i$(_,_,p,_), LinkGen*ParentChild*$_j$(_,_,p,_).
- Define a new query queryDisjoint with the disjunctions of the queryDisjoint*GSName*$_k$ queries previously defined for such a GS:
queryDisjoint:= queryDisjoint*GSName*$_1$ | queryDisjoint*GSName*$_2$ |...| queryDisjoint*GSName*$_n$.
If there are only two subclasses, a single queryDisjoint*GSName*$_k$ is defined which is directly assigned to queryDisjoint.
- Include the negation (!) of queryDisjoint in the conforms query.
Example of use
queryComplete:= p is InstanceEmployee, fail LinkGenEmployeeFullTimeEmployee(_,_,p,_),
fail LinkGenEmployeePartialTimeEmployee(_,_,p,_).
queryDisjointGS1:= LinkGenFullTimeEmployee(_,_,p,_), LinkGenPartialTimeEmployee(_,_,p,_).
queryDisjoint := queryDisjointGS1.
conforms := !queryComplete & !queryDisjoint.

representing the opposite of the semantics given by the *complete* partition, that is, that every instance of a general classifier (*Parent*) is not an instance of any of its specific classifiers (*Child*$_1$, *Child*$_2$, ..., *Child*$_n$). Finally, for all generalization sets with *complete* constraints, the negation of the `queryComplete` query is included once in the final `conforms` query for verifying the *complete* semantics.

Similarly, for each generalization set with the *disjoint* constraint we need to make sure that the specific classifiers (*Child*$_1$, *Child*$_2$, ..., *Child*$_n$) cannot share common instances, that is, they cannot correspond to the same general classifier (*Parent*). In order to represent this fact, we make sure that there is not a pair of instances of specific classifiers (*Child*$_i$, *Child*$_j$) corresponding to the same instance of the general classifier (*Parent*). More specifically, we firstly define a query `queryDisjoint`*GSName*$_k$ (being *GSName* the name of the generalization set), for each disjoint pair of subclasses, representing the fact that they can share common instances (see Table 4). Later, we define another query `queryDisjoint` with the disjunctions of the `queryDisjoint`*GSName*$_k$ queries previously defined, in order to represent the opposite of the *disjoint* constraint, that is, that there can be instances of different subclasses which share the same subclass instance. Finally, we include the negation of the `queryDisjoint` query in the `conforms` query for verifying the *disjoint* constraint semantics. We have to note that when there are only two subclasses in the generalization set a single `queryDisjoint`*GSName*$_k$ is defined which is directly assigned to `queryDisjoint`. Additionally, when the generalization set is *disjoint* and there is only a specific classifier, the verification of the negation of the `queryDisjoint` query would represent the fact that there are no instances of the corresponding `LinkGen`*ParentChild* data type, which would make no sense. For this reason, this query is not defined when the generalization set has only a specific classifier.

Since the *incomplete* and *overlapping* partitions represent the opposite, less restrictive constraints to the *complete* and *disjoint* partitions, respectively, to represent such partitions, we only have to omit the definition of the `queryComplete` and `queryDisjoint` queries, respectively. As an example, in Table 4 we show the two queries defined for the {`complete, disjoint`} generalization set defined in the case study for the specialization of the "Employee" class.

Finally, the Formula expressions defined in *M1b* constitute an FD called $InstanceLevel_{FD}$. This domain extends the $MetaLevel_{FD}$ domain, defined at level M2 (see Figure 4), since it creates new Formula data types on the ones defined in $MetaLevel_{FD}$.

4.3. Level M0.

Finally, to allow Formula to reason and search for valid instances of the specific classes, associations, properties, and generalizations of the source CD, we include the `Introduce(f,n)` command, with the corresponding `Instance`, `Link`, `LinkGen`, or `Slot` data types, as `f`, and a specific number as `n`, to indicate the number of valid instances of such data types that we desire For-

Table 5: Excerpt of the proposal regarding M0 level.

M0 level	
Class	**Association**
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,star, Class("Person",false),0,2), InstancePerson(93, Class(''Person'',false)), InstancePerson(96, Class(''Person'',false)))
Property	**Generalization**
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)), StringProperty("name","",1,1),202) namePersonSlot(InstancePerson(96,Class("Person",false)), StringProperty("name","",1,1),201)	*Formula instructions pattern:* **[Introduce(LinkGen**g.general.name**+**g.specific.name**,** number**)]** *Example:* [Introduce(LinkGenPersonEmployee, 2)] *Example of the Formula generated instances:* LinkGenPersonEmployee(67, Generalization(Class("Person",false), Class("Employee",false), InstancePerson(93, Class(''Person'',false)), InstanceEmployee(56, Class(''Employee'',false)))

mula to generate as part of the resulting OD. For example, we define the `[Introduce(InstancePerson, 2)]` command so that Formula searches two valid instances of `InstancePerson` (see Table 5).

At this point a special remark has to be made regarding the value of the bound `n` parameter to provide to each `Introduce` command. More specifically, there is a dependency among the instances in the resulting model (for example, an instance of the data type `InstanceFullTimeEmployee` requires an instance of `InstanceEmployee` and `InstancePerson`). In particular, the maximum number of the *property* slots would depend on the bound chosen for the corresponding owner class (normally, both bounds are the same unless the property multiplicity is more than one). Special attention is required for *generalization* relationships, since there are delicate dependencies among instances of parents and children. On one hand, if the generalization set is *complete* and *disjoint*, the more suitable value for the bound of the parent would be the sum of the bounds of its children. If the generalization set is *complete* and *overlapping*, to consider the more restrictive option in which specific classifiers do not share common instances, we can choose again the sum of the bounds of the children as the parent bound. On the other hand, if the generalization set is *incomplete*, where there could be some instances of the general classifier that could not be classified as any of the specific classifiers from the generalization set, the bound of the parent, both in *overlapping* and *disjoint* cases, could be set to the sum of the bounds of its children plus the number of specific instances of the parent we want Formula to generate. Additionally, the bounds of the `LinkGen`*`ParentChild`* instances would have to be chosen according to the previous decisions, coinciding with the bounds of the instances of the specific child. Finally, extra calculations would have to be completed with the bounds of the *association* links, taking into account the corresponding multiplicities and the bounds chosen for the instances of the associated classes.

Finally, the Formula expressions defined at this level are included in a partial model, we have called $CDInstance_{FPM}$.

We want to note that, although our conceptual proposal requires the user to provide bounds for classes, properties, associations and generalizations, as we will explain later, by using the CD2Formula plug-in, the user would have to manually indicate the number of valid instances per class, property and association. However, the bounds of the `LinkGen`*`ParentChild`* instances would be directly provided by the plug-in according to the bounds of the instances of the specific child.

4.4. Some remarks regarding other Class Diagrams' elements

A special remark has to be made regarding the *Classifier* metamodel element, *association classes*, *strong composition*, and *user–defined data types.* On one hand, the *Classifier* element is defined by a derived data type, as the union of the `Class` and `Association` primitive data types so that we can generally refer to classes and associations. On the other hand, *association classes* are translated in the same way as associations, so that they can register the associated classes, but with the particularity of owning properties. The owned properties are established in the associated slots data types at the M1b level,

Table 6: Strong composition translation and example of use.

Translation rule

- Per each *Part* included in the composite aggregations with whole*1*, *whole*$_2$, ..., whole$_n$ respectively, we define the query:

```
queryComposition:=p is InstancePartName,
                             count(LinkassocPartWhole1(_,_,p,_))+
                             count(LinkassocPartWhole2(_,_,p,_))+...+
                             count(LinkassocPartWholen(_,_,p,_))!=1.
```

- Include its negation (!) in the `conforms` query.

Example of use

```
queryComposition:=p is InstanceDepartment,
                             count(LinkmakesUpOf(_,_,p,_))!=1.
conforms:=!queryComposition & other queries.
```

thanks to the definition of the `Element` data type. This data type is defined as the union of `Instance` and `Link` data types, and later, established as the type of the slot's owner element (see the translation of properties at level *M1b* in Figure 3). For example, for the translation of the "Assigned" association class, we define the element: `Assigned is Association(''Assigned'', classDepartment,1,1,classEmployee,3,star)` included in level *M1a* and the term `primitive LinkAssigned::= (id:Integer, type:Association, department: InstanceDepartment, memberOf:InstanceEmployee)` included in level *M1b*. Additionally, its property `date` is translated by defining the elements `dateAssigned is DateProperty(''date'', '''',1,1)` and `HasProperty(Assigned, dateAssigned)` included in level *M1a* and the term `primitive dateAssignedSlot::= (owner:Element, prop:DateProperty, value: String)` included in level *M1b*.

Regarding *composite aggregation* (also known as *strong composition*), we have taken into account as a special form of association that it can be refactored as binary associations together with additional OCL constraints (see [20]). For this reason, we also represent the strong composition element as associations, defining additional Formula queries imposing such OCL constraints. More specifically, since a composition represents a strong form of whole/part association and requires a part instance to be included in at most one composite at a time [7, 20], for each part included in strong composition relationships, we define in the *InstanceLevel*$_{FD}$ domain an additional Formula query representing the fact that a part instance is included in more than one composite at a time (see in Table 6 the definition of the query `queryComposition`). Finally, to verify the dependence of a part with at most one whole each time, we define the `conforms` query with the negation of the query previously defined as shown in Table 6. In this way, we note that we are able to represent both (1) compositions where there is only one whole (the multiplicity on the whole end of the composition is exactly 1) and (2) compositions where several whole classes point to the same part class but the same part instance cannot be used simultaneously in different whole classes (the multiplicity on the whole end of the composition is 0..1). As an example of the first case, in Table 6 we show the query defined to specify the additional semantics of the strong composition on Figure 1 between `Company` and `Department`, given by the association `makesUpOf`. Additionally, the negation of this query is included in the `conforms` query.

As for *user–defined data types*, we create a specific domain called *User-DataTypes*, which is extended by the *MetaLevel*$_{FD}$ domain (see line 1 in Figure 3), so that it can use the data types defined by the user. For each property defined by the user, this specific domain creates a primitive Formula type following the translation rules of properties defined at the M2 level. In the particular case of *enumeration* data types, we define a Formula constant with the list of possible fixed values within curly brackets (for example, in the case of the `Gender` enumeration type in the case study, we define the constant `Gender::= {female,male}`). In order to be able to define properties of the enumeration data type, we also define a primitive data type as described before in the translation of properties at the M2 level (in the previous example, we would define the data type `GenderProperty` as `GenderProperty::= (name: String, def: Gender, lower: Natural, upper: UpperBound)`.

5. Translation of Class Diagram constraints

Among the different constraints that can be applied to a CD, we can distinguish those predefined in UML and defined on the metamodel, from those user-defined that are defined on the specific CD [2]. On one hand, the first are used in the UML semantics description to define the *well–formedness rules* for UML models, which ensure that the UML model is consistent with the UML metamodel. The *user–defined constraints*, on the other hand, are used to impose the otherwise unrestricted particularities intrinsic to the particular CD. Both types of constraints can be specified using a natural description and may be followed by a formal constraint expressed in OCL [3]. Additionally, the user–defined constraints may be implicit in the model notation (like the multiplicity constraints in UML associations) or explicitly established using OCL. Regarding the OCL representation, the well-formedness rules are defined in terms of a set of OCL *invariants* for each UML metaclass [21], where user–defined constraints can be also defined in terms of OCL *preconditions* and *postconditions*. Taking this into account, we focus on invariants and describe how class invariants, specified within the chosen OCL fragment, are translated to Formula.

5.1. Overview of our approach for translating constraints

As presented previously, the OCL integrity constraints are known to be undecidable [5, 6]. Such undecidability has been tackled in literature by defining methods that allow UML/OCL reasoning at some level. Examples of such methods are [5, 22]: (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 subset of OCL, which corresponds to the OCL constraints defined using the fragment of OCL presented in Figure 5, and provides the translation of this fragment to the Formula tool for OCL constraints' decidable reasoning. In this section, we show that the proposed fragment of OCL can be formally encoded in Formula;

```
OCLExpr       ≡ LiteralExpr | RelExpr | MulOrAddExpr | not OCLExpr |
                OCLExpr1 and OCLExpr2 | OCLExpr1 or OCLExpr2 | Path |
                Path SelectOpe | Path BooleanOpe |
                Path UnionOpe | Path CollectOpe
LiteralExpr   ≡ Integer | Real | String | true | false
RelExpr       ≡ OCLExpr relOpe OCLExpr
relOpe        ≡ < | <= | > | >= | = | !=
MulOrAddExpr  ≡ OCLExpr ope OCLExpr
ope           ≡ +| - | / | *
Path          ≡ PathItem | PathItem.Path
PathItem      ≡ role | classAttribute | operation | classRoleName.role |
                classRoleName.classAttribute | classRoleName.operation |
                classRoleName.transClosOperator
SelectOpe     ≡ -> select(OCLExpr)BooleanOpe| ->select(OCLExpr)SelectOpe
BooleanOpe    ≡ -> size()| -> forAll(OCLExpr)
UnionOpe      ≡ -> union(Path)
CollectOpe    ≡ -> collect(OCLExpr)BooleanOpe| ->collect(OCLExpr)SelectOpe
                -> collect(OCLExpr)UnionOpe
```

Figure 5: Syntax of the OCL fragment.

thus, we allow finite reasoning for every CD constraint expressed in OCL and defined with the constructors considered in our OCL fragment.

Next, we introduce the chosen OCL fragment and explain our approach for translating it. To provide the reader a better idea of this translating approach, first we 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.

5.2. OCL fragment

We consider the OCL invariant `context C inv: expr(self)`, where `C` is the class in the 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 role names in associations (such as `p.children`, `p` being an instance of `Person` in Figure 1), attribute names (such as `c.name`, `c` being an instance of `Company`), or operations (for example, `c.hireEmployee(e)` an operation defined in the `Company` class). Taking this into account, in Figure 5 we represent the syntax of our specific fragment. As it can be seen, `OCLExpr` is defined recursively. For example, an `OCLExpr` can be the result of applying relational operations to other `OCLExpr` expressions (e.g. `OCLExpr` $<$ `OCLExpr`). Additionally, an `OCLExpr` can be the result of applying a boolean operation `BooleanOpe` to a `Path` or a `Path` to which a `SelectOpe`, a `UnionOpe`, or a `CollectOpe` is applied. An `OCLExpr` can be a `Path` expression which represents the structural method of defining navigation paths, starting from a `PathItem`, by combining roles' names, class attributes' names, or operations (including the `transitiveClosure`), with the dot ('.') operator. Also, primitive literal expressions are considered (identified as `LiteralExpr`) to allow including constant values such as `true`, `1.5` or "`text`". For an explanation of the semantics of OCL, we refer to [3].

5.3. OCL invariants

Formula does not have a concept similar to that of OCL invariants but enables the possibility of defining queries, which provide a method to represent invariant semantics. As an 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`.

Example 1. We explain this rule by applying it to the user–defined OCL constraint presented in Table 7, which is defined for the CD of Figure 1. This OCL invariant formalizes the constraint "The people registered in the system must be older than 18 years old [1]." Next, we will explain each step of our proposal by applying it to this particular invariant.

First–step. This step is carried out by 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 7). First order logic states that the universal quantifier corresponds to a negated existential, so the previous expression is equivalent to the one labeled with (1'), where *`FOL`*`(not expr(self))` corresponds to the mapping of `not expr(self)` into FOL.

[1]Although OCL defines an invariant to be true for all instances of the classifier and can be represented using the forAll OCL operation [3], we adopt the reduced version.

Table 7: Invariant translation 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 1	
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)

Example 1. The invariant of our example can be represented in FOL as the expression labeled by (1) in Table 7 or equivalently, by (1').

Second–step. Each constraint logic program P can be translated into FOL according to its *Clark Completion* P^* [23]. 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, from the FOL representation of P (P^*) to P, can be carried out by applying inverse versions of the *Clark Completion* algorithm [24], which compile the 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 CD, and in our approach such instances are defined by the data types defined in the $CDInstance_{FPM}$ partial model, such Formula expressions are written in terms of the `Instance`*className*, `Link`*associationName*, `LinkGen`*g.general.name+g.specific.name*, and/or *proper-*

tyName+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))` using the corresponding Formula terms and predicate symbols $\in InstanceLevel_{FD}$, and Formula constraints, in such a way that the resulting expression is evaluated to `true` (see step labeled (2)).

Example 1. The application of this second step to our example consists of representing, in terms of Formula expressions, the elements necessary to navigate from the context class `Person` to the fact `age(self)<18` , as presented in Table 7 with label (2). In such expressions `self` $\in$ `InstancePerson` corresponds to the result of the application of the function $FOL^*()$ to the context class `Person`, and the rest of the terms correspond to the application of the function $FOL^*()$ to `age(self) <18`, that is, considering type expressions as necessary to reach, from `InstancePerson` to the `agePersonSlot` slot whose `value` 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 represented by the definition of a `query` and the verification of its negation in the `conforms` query (see step labeled (3) in Table 7). This step is materialized by the application of the function $CLP()$, which basically rewrites the terms resulting from (2), and joins them by ‘,’, omitting the translation of the expression `self` $\in$ `Instance`*C.name* since in ***FOL****`(`***FOL*** `(not expr(self)))` the field `self` unequivocally corresponds to `Instance`*C.name*.

Example 1. The application of function $CLP()$ would result in the Formula expression presented in Table 7. Note that since in `agePersonSlot(self,_,val)`, the field `self` corresponds to `InstancePerson`, it has been omitted from the definition of the `query`.

To sum up, the translation of an invariant is carried out by the composition of the three defined functions ***CLP***$\circ$***FOL***$^*\circ$***FOL*** `()`. In order to improve the readability and understandability of the translation expressions, from now on we use the function ***Trans*** `()` defined as that composition.

Table 8: Translation of Boolean OCL expressions

OCL expression	Translation approach
`expr1 and expr2`	*`Trans`*`(expr1) & `*`Trans`*`(expr2)`
`expr1 or expr2`	*`Trans`*`(expr1) | `*`Trans`*`(expr2)`
`not expr`	*`Trans`*`(not expr)`

5.4. Boolean OCL expressions

Having presented our approach for the translation of a simple OCL invariant, next we are going to describe the translation of the rest of the OCL expressions included in our OCL fragment. More specifically, in Table 8 we present the translation of the *conjunction*, *disjunction*, and the *negation* operators of the OCL fragment, which we consider easily understood taking into account our previous explanations. As an example, we briefly explain the translation of an OCL expression with the *conjunction* operator `expr1 and expr2`. In particular, if *`Trans`*`(expr1)` results in the verification of a query `!query1` in the `conforms` one, and *`Trans`*`(expr2)` 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`). Finally, we represent this translation rule as *`Trans`*`(expr1)` & *`Trans`*`(expr2)`, where each expression is translated recursively using the translation rules presented in the remainder of this paper by applying the function *`Trans`*`()`.

5.5. OCL collections

The translation of OCL expressions, which include operations in collections (that is `CollectOpe`, `UnionOpe`, `SelectOpe` and `BooleanOpe` expressions) and the particular case of operations implementing a *transitive closure* of a relationship, require extra attention. Since these expressions work with collections, a special remark must be made. The OCL standard defines a number of collection constructs, such as *Sets* (unordered without duplicates), *Bags* (unordered and may contain duplicates), *OrderedSets* and *Sequences* (ordered that may contain duplicates) [3](Sec. 11.6). We can infer from the Formula documentation [12] that the Formula language works with unordered collections (sets and bags).

Among the predefined operations on collections provided by the OCL standard [3], our OCL fragment considers: `forAll`, `size`, `select`, `union`, and `collect`. In the next subsections, we provide a complete explanation of the translation of the `select` operation and describe how operations implementing a *transitive closure* are represented in Formula. We also give a description of the translation of the remainder operations, not provided in [16].

5.5.1. ForAll, Size, Select, Union, and Collect operations

forAll. On one hand, the general syntax of OCL universal quantifier expressions [3](p. 29, Sec. 7.7.3) is `C -> forAll(c|expr(c))`, where `expr(c)` refers to a boolean expression, which is evaluated for every item in the collection `C`. If the boolean expression `expr(c)` is `true` for all possible instantiations `c` in the collection `C`, the whole expression is `true`, otherwise the expression is `false`. As explained before, Formula does not have a *universal quantifier* but we can represent it by verifying the negation of a query representing the opposite of the constraint in the universal quantifier (see in Table 9 the rule for translating the `forAll` operator).

Example 2. As an example of the translation of this operator, we demonstrate the translation of the OCL invariant labeled (2) in Figure 1, which formalizes the constraint "The salary of an employee must be greater than 1000." Following our approach, this constraint is represented in Formula by following our three-step process, which is presented in Table 9. In particular, for the translation of this constraint, to represent instances of the `WorksIn` association, we have used the Formula data type `LinkWorksIn(id,assoType,employer,employee)`, which uses the identifier, the type of association, the corresponding `InstanceEmployee` as employee, and the `InstanceCompany` as employer as fields.

size. On the other hand, the `size` operation is applied on a collection, returning the collection's cardinality (i.e., the number of elements in the collection). The

Table 9: Translation of the `ForAll` operation.

Translation of the `forAll` operation:	`C -> forAll(c|expr(c))`
`query:=`*CLP*`(`*FOL**`(`*FOL*`(not expr(c))))`. `conforms:= !query`.	
Example 2	
`context Company inv: self.employee->forAll(e:Employee|e.salary>1000)`	
First-step:	∀ `self` ∈ `Company em` ∈ `employee(self) salary(e)` >=`1000`. (1) ¬(∃ `self` ∈ `Company em` ∈ `employee(self) salary(em)<1000)`. (1')

Second-step:	¬(∃ `self` ∈ `InstanceCompany(id,type)`
	`wi` ∈ `LinkWorksIn(id,type,self,em)`
	`em` ∈ `InstanceEmployee(id,type)`
	`salaryEmployeeSlot(em,def,val) val<1000 )` (2)

Third-step:	`query:= LinkWorksIn(_,_,c,em),`
	`salaryEmployeeSlot(em,_,value), value<1000.`
	`conforms := !query.` (3)

syntax of this operator is `C -> size()` and can only be applied to countable sets. Formula has a `count` operator, which can be used to count instances of a specific term given as parameter, so intuitively, we use this operator to translate the `size` operation, as presented in Table 10.

select. Regarding the `select` operation, it is used to select members of a collection that satisfy a boolean expression and return a new collection that contains only those members [3] (27, Sec. 7.1.1). The OCL syntax of this operation is `C -> select(c|expr(c))`. This statement selects the elements of the collection `C`, which satisfy the boolean expression `expr(c)` and returns a new collection with only those elements. Using Table 10, next we describe the translation of this operator, applying it to a constraint in which the `size` operation also appears (in this way, we also provide an example of an application for the `size` operation).

Example 3. Let's consider the translation of the OCL invariant labeled (3) in Figure 1 also presented in Table 10, which represents the fact that "in a company, at least two of the employees are female."

First–step. In order to obtain a subcollection from a set of elements, based

Table 10: Translation of the `size` and `select` operations.

Translation of the `size` operation: `C-> size()`	
`count(`*CLP*`(`*FOL**`(`*FOL*`(C))))`.	
Translation of the `select` operation: `C-> select(c|expr(c))`	
$\mathtt{S}_{C,expr}$`Type::=(self:T`$_{self}$`,sele:T`$_{sele}$`).` $\mathtt{S}_{C,expr}$`Type(self,sele):-`*CLP*`(`*FOL**`(`*FOL*`(expr(c))))`	
Example 3, including the `size` and `select` operations	

`context Company inv:  self.employee ->`
 `select(p:Person| p.gender=Gender::female)-> size()>=2`	

First-step:	- `∀(s ∈ S`$_{C,expr}$` ↔ self ∈ Company s ∈ employee(self) &` `gender(s) = Gender::female)` - `size(S`$_{C,expr}$`)>=2`, or equivalently, `¬(size(S`$_{C,expr}$`)<2)` (1)
Second-step:	- `∀( (s,self) ∈ FemaleEmp(x,y) ↔ LinkWorksIn(_,_,self,s)` `genderPersonSlot(s,_,val) val= female.` - `¬( size(FemaleEmp(x,y))<2 )` (2)

Third-step:	- `FemaleEmp::= (self:InstanceCompany,s:InstanceEmployee).`
	 `FemaleEmp(self,s):- LinkWorksIn(_,_,self,s),`
	 `LinkGenPersonEmployee(_,_,p,s),`
	 `genderPersonSlot(p,_,val), val= female.`
	 Finally, the translation of the constraint would be:
	 `query:= self is InstanceCompany, count(FemaleEmp(self,_))<2.`
	 `conforms := !query.` (3)

on [25], we propose defining a new symbol $S_{C,expr}$. The idea is that this symbol represents the members in the collection we want to select from the source collection. The symbol $S_{C,expr}$ is defined as:

$$\forall[\mathtt{s} \in \mathtt{S}_{C,expr} \leftrightarrow \mathtt{s} \in \mathit{FOL}(\mathtt{C})\ \&\ \mathit{FOL}(\mathtt{expr(c)})](1)$$

Example 3. The application of this first step to the example corresponds to the definition of the symbol $S_{C,expr}$ as presented in Table 10. Note also the translation of the first step of the `size` operation, applied to the new symbol representing the subcollection of elements we want to count.

Second–step. We rewrite the symbol $S_{C,expr}$ in terms of a new Formula data type $\mathtt{S}_{C,expr}$`Type`, by applying the function *FOL**($\mathtt{S}_{C,expr}$`Type`), as follows:

$$\forall[\mathtt{s} \in \mathtt{S}_{C,expr}\mathtt{Type} \leftrightarrow \mathtt{s} \in \mathit{FOL}^*(\mathit{FOL}(\mathtt{C}))\ \&\ \mathit{FOL}^*(\mathit{FOL}(\mathtt{expr(c)}))](2)$$

Example 3. The application of the second step is the redefinition of the type $\mathtt{S}_{C,expr}$`Type`, to a new Formula data type, which we have called `FemaleEmp`, as depicted in Table 10. In particular, with the compound term `LinkWorksIn(_,_,s, self)` we refer to the instances that correspond to the contracts of the employee `s` with the company `self`. With `LinkGenPersonEmployee(_,_,p,s)` we refer to the specialization relationship between the instance employee `s` and the instance person `p`, while with the compound term `genderPersonSlot(p,_,val)`, we take the gender slot of the person `p` corresponding to the employee `s`. Finally, with `val= female`, we select the female persons from these employee instances.

Third–step. In this step, we need to define the new data type $\mathtt{S}_{C,expr}\mathtt{Type}$ and populate it with the elements of the collection we want to select. More specifically, first, we define a new type $\mathtt{S}_{C,expr}\mathtt{Type}$ as:

$$\mathtt{S}_{C,expr}\mathtt{Type\ ::=\ (self:\ \ T}_{self}\mathtt{,\ sele:\ \ T}_{sele}\mathtt{)}$$

where $\mathtt{T}_{self}$ is a Formula type expression corresponding to the Formula type expression of the invariant's context, that is, to `Instance`*C.name*, and where $\mathtt{T}_{sele}$ is a Formula type expression corresponding to the Formula type expression representing `expr(c)`.

Second, we create a *production* rule that generates new entries in the fact-base of Formula, populating the previously defined type with facts representing just the members in the collection we want to select:

$$\mathtt{S}_{C,expr}\mathtt{Type(self,sele):-}\ \mathit{CLP}(\mathit{FOL}^{*}_{CT}(\mathit{FOL}(\mathtt{C}))),\ \mathit{CLP}(\mathit{FOL}^{*}_{CT}(\mathit{FOL}(\mathtt{expr(c)})))$$

Example 3. In this step we define the `FemaleEmp` Formula type as presented in Table 10 and create the rule `FemaleEmp(self,s)`, which populates the new data type with pairs *company-employee* in which the employees are females. Finally, the overall constraint is translated as described in Table 10.

union. The `union` operation is used to join collections [3] (23, Sec. 7.6.11). The OCL syntax of this operation is `collection1 -> union(collection2)`. Our proposal for the translation of this operation is similar to the translation of the `select` operation in the sense of that it is mainly based on the semantics of the Formula production rules. In particular, we create a new Formula data type that represents the elements in each collection (which are elements of the same nature). Later, as in the case of the `select` operation, we create a production rule per each collection (`collection1` and `collection2`), which would populate the new data type with the elements included in the corresponding collection. Finally, it results in a collection of facts representing the union of the initial collections. An example of the translation of the `union` operation could be seen in the next subsection, where we discuss *transitive closure*.

collect. The `collect` operation is used to specify a collection that is derived from some other collection, which contains different objects from the original collection [3] (28, Sec. 7.7.2). The OCL syntax of this operation is `C -> collect(c|expr(c))`. This statement returns the collection of the results of all the evaluations of `expr(c)`. Related with the use of the `collect` operation is the consideration of collections of collections in OCL constraints. Based on the OCL specification, automatic flattening is carried out when using the `collect` operation. Similarly, implicit flattening is considered when used with the shorthand notation for `collect` (see subsection 7.7.2 "collect" operation in [3], p. 29). Since the `collectNested` operation returns a nested collection, the `flatten` operator must be explicitly applied to get the flattened version (see subsection 11.9.1.6 "collect" in [3], p. 170). At this point, we have to note that Formula does not support collections of collections [12]. For this reason, we provide a proposal to be used for the translation of both the `collect` operation and the flattened collections represented by using the shorthand notation for `collect`.

Our approach consists of (1) defining a Formula data type representing the members in the collection (or flattened collection), (2) creating a production rule used to populate the new data type with facts that represent just the members in the collection (or flattened collection), and (3) using such a data type in the translation of the remainder constraint. As an example, in Table 11 we present the translation of two different constraints where the shorthand notation for `collect` is used. In particular, the first constraint represents the fact that "for each department, all employees earn more than 1000", while the second one

Table 11: Examples of translation of flattened collections.

Example 1. `context Department inv: self.project.participant -> forAll(e: Employee| e.salary>1000)`

```
employeesDepartment ::= (d:InstanceDepartment, e: InstanceEmployee)..
employeesDepartment(d,e):- Linkcontrols(_,_,p,d), LinkworksOn(_,_,p,e).
queryFlattened := employeesDepartment(d,e),
                            salaryEmployeeSlot(e,_,value), value<=1000.
conforms := !queryFlattened.
```

Example 2. `context Department inv: self.project.participant -> size()=6`

```
employeesDepartment ::= (d: InstanceDepartment, e:InstanceEmployee).
employeesDepartment(d,e) :- Linkcontrols(_,_,p,d), LinkworksOn(_,_,p,e).
queryFlattened2 := count(employeesDepartment(d,_))!=6
conforms := !queryFlattened2.
```

states that "the total of employees that participate in the projects controlled by a department must be 6". We note that in both constraints, `self.project` delivers a `Set(Project)` and `self.project.participant` delivers a `Set(Set(Employee))`, which would result in a `Set(Employee)`. We would like also to note that, although in these examples we consider the shorthand notation for the *collect* operation, the same translation idea could be used equivalently for the translation of the `collect` operation.

5.5.2. Transitive closure

Transitive closure is normally needed to represent model properties, which are defined recursively. Additionally, it is often used in reasoning about partial orders, and thus widely found in modeling applications. The translation of closures is not straightforward since, on one hand, they are not finitely axiomatizable in FOL and, on the other hand, OCL also does not support them natively [26, 27]. Nevertheless, it is possible to define the transitive closure of relations, which are known to be finite and acyclic.

Let's consider the OCL operator labeled (4) in Figure 1, defined in the context `Person`, where `ancestors` are recursively defined, in order to represent the transitive closure of the relation defined by `parents`. The expression `self.parents` denotes the set of all direct supertypes, whereas `self.ancestors` denotes the transitive closure of direct supertypes.

The transitive closure is illustrated in [26], which defines `ancestors` by:

$$\texttt{APar(x) = Par(x)} \cup \{\texttt{y} \mid \exists \texttt{ z} \in \texttt{Par(x)} \wedge \texttt{y} \in \texttt{APar(z)}\}$$

where `Par(x)` and `APar(x)` are the translations of `x.parents` (note that the OCL variable `self` has been substituted by x, a more common variable in mathematics) and `x.ancestors`, respectively.

As stated in [26], this definition can be expressed in FOL by the formula:

$$\texttt{r}^*\texttt{(x,y)} \leftrightarrow \texttt{r(x,y)} \vee (\exists \texttt{ z r(x,z)} \wedge \texttt{r}^*\texttt{(z,y))}$$

where `Par` and `APar` are substituted for by the relation symbols `r` and $\texttt{r}^*$ with `r(x,y)` denoting $\texttt{y} \in \texttt{Par(x)}$, and `r*(x,y)` denoting $\texttt{y} \in \texttt{APar(x)}$.

This formula is interpreted by the structure (U,R,R^*) where U is the universe, and R and R^* are interpretations of the relations `r` and $\texttt{r}^*$, respectively. The author in [26] presents countermodels for this formula whereby R^* does not coincide with the transitive closure of R; however, the author states that if (U,R,R^*) is a finite model and the axiom $\neg\texttt{r}^*\texttt{(x,x)}$ holds (that is, R^* is enforced to be acyclic), then R^* is a correct definition of transitive closure.

Moreover, acyclicity constraints are easily captured in CLP since it exposes fixpoint operators via recursive rules [14, 18]. In particular, we represent transitive closure in Formula using recursive rules as the following:

$$\texttt{r}^*\texttt{(x, y) :- r(x, y).}$$
$$\texttt{r}^*\texttt{(x, y) :- r(x, z), r}^*\texttt{(z, y).}$$

where the first expression encodes $\texttt{r}^*\texttt{(x,y)} \leftrightarrow \texttt{r(x,y)}$, and the second expression encodes $\exists \texttt{ z r(x,z)} \wedge \texttt{r}^*\texttt{(z,y)}$. Based on the translation rules we have defined previously, next we explain the translation to Formula of an OCL constraint with an operation referring to a transitive closure relationship.

Example 4. Let's consider the OCL constraint labeled (4) in Figure 1, which formalizes the constraint "A person cannot be married with an ancestor." This invariant is represented in OCL as `context Person inv: self.ancestors ->`

`forAll(p | p!= self.spouse1)`, where the operator `ancestors` is defined as presented previously. The representation in Formula of the `ancestors` operator of our case study would consist of: (1) the definition of a new Formula type `ancestors` that represents pairs `child-parent`, together with (2) two production rules that populate such a type (we note that such a translation could be also easily inferred following the translation rules of the `union` and the `collect` operations, presented in the `ancestors` definition):

```
ancestors ::= (child:InstancePerson, parent:InstancePerson).
ancestors(x, y) :- LinkFamily(_,family,x,y).
ancestors(x, y) :- LinkFamily(_,family,x,z),  ancestors(z,y).
```

Since the OCL invariant has a `forAll` operator, we can use its translation rule, resulting in the expression:

```
query:= ancestors(child,parent),
        LinkMarriage(_,_,spouse1,spouse2),
        spouse1=child, spouse2=parent.
conforms:= !query.
```

where `LinkMarriage(id,assoType,spouse1,spouse2)` corresponds to the translation of the association `marriage`.

As advanced previously, since the translation of the OCL constraints in a CD are defined in terms of the data types created in the $\mathit{CDInstance}_{FPM}$ partial model, the Formula expressions resulting from the translation of the OCL constraints are included in such a partial model.

5.6. Other issues regarding OCL

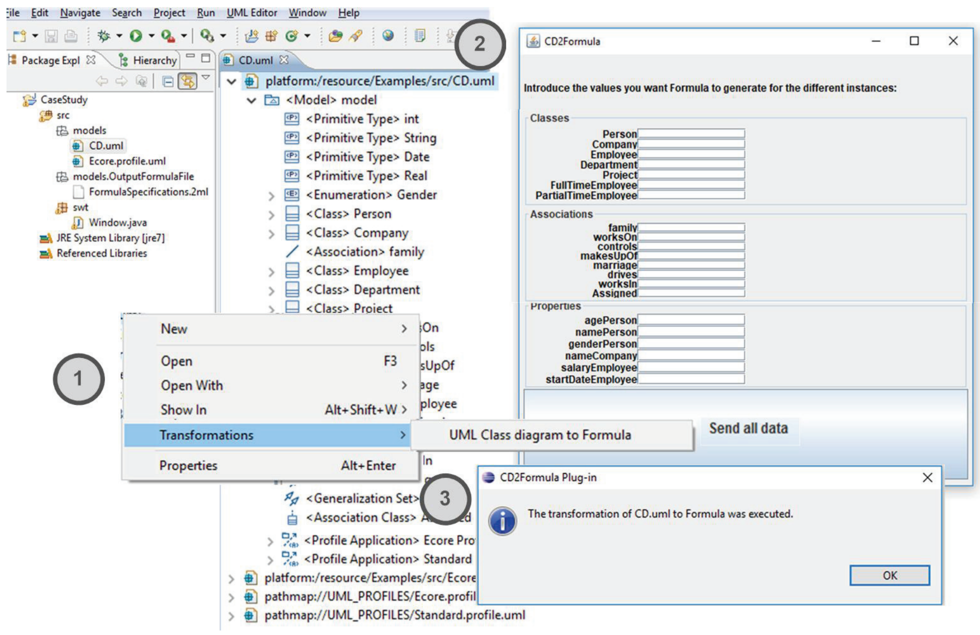
There are several OCL operations and expressions whose representation in Formula is straightforward by applying equivalences and using our translation proposal of the chosen OCL fragment (see in Table 12 these operations and expressions and their equivalences considering [28]). This is the case of, for example: (1) the *exclusive disjunction* operator (`xor`) [3] (p. 153, Sec. 11.5.4), which is easily translated considering the conjunction/disjunction and negation

Table 12: Other OCL operators and expressions, and their equivalences

OCL operation and expressions	Equivalent expression
`expr1 xor expr2`	`(expr1 or expr2) and not (expr1 and expr2)`
`C → reject(c| expr(c))`	`C → select(c| not expr(c))`
`C → isEmpty()`	`C → size()= 0`
`C → notEmpty()`	`C → size()> 0`
`expr1 implies expr2`	`not expr1 or expr2`

operations, (2) the `reject` operator [3] (p. 27, Sec. 7.7.1), easily translated using its equivalence with the `select` operator, (3) the `isEmpty`/`notEmpty` operators [3] (p. 157, Sec. 11.7.1), whose translation to Formula is easily performed using the `size` translation, (4) the `exists` operator [3] (p. 30, Section 7.7.4), which is easily inferred from the translation of the `forAll` operator considering the *existential quantifier* character of queries, and (5) the `implies` operator [3] (p. 154, Sec. 11.5.4), which is easily translated considering the disjunction and negation operations.

An additional remark has to be made regarding the predefined OCL properties that apply to all objects [3], such as `oclIsTypeOf, oclIsKindOf`, and `oclAsType` (pp. 22, Sec. 7.6.9). In particular, although the mismatch among instances and types is checked by the Formula tool, to our knowledge, it does not provide specific operations which allow to represent directly the previous OCL operations. This similarly happens with the operation `oclIsUndefined`, or with OCL operations that are state dependent. On the one hand, we do not give support to the `oclIsUndefined` operation, thus our proposal does not implement four-valued OCL logic. On the other hand, extra attention must be given to OCL operations that are state dependent (such as the operation `oclIsInState`, which evaluates whether the object is in a specific state, and `oclIsNew`, which checks whether the object does not exist in the previous state of the system but exists in the current state). More specifically, in OCL operations that are state dependent, a UML state machine diagram is required and representing UML state machines in Formula is out of the scope of this work, but we consider that defining a proposal for reasoning about UML dynamic diagrams constitutes an interesting issue for further work.



Introduce the values you want Formula to generate for the different instances:

The transformation of CD.uml to Formula was executed.

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

To sum up, we have shown that the proposed fragment of OCL can be formally encoded in Formula; thus, we can reason about CD constraints expressed using the constructors of the considered fragment of OCL. In particular, we are able to represent in Formula both well–formedness rules and user–defined constraints (implicit in the model and OCL explicit constraints), specified with the constructors considered in our OCL fragment. In particular, well–formedness rules are represented in Formula and included in the $MetaLevel_{FD}$ domain. Regarding user–defined constraints implicit in the model notation (like multiplicity constraints in associations), they are also represented as shown in this section, since they can also expressed directly as OCL constraints.

6. Tool support and reasoning process of the case study

In this section, we describe the development aspects of the *CD2Formula* Eclipse plug–in, which allows us to automatically perform the transformation from class diagrams to Formula. Also, in order to illustrate the usefulness of our approach, we apply it to our case study.

6.1. Development of the CD2Formula Eclipse plug–in

The first step in our overall process consists of translating the model, which we want to reason about, into the Formula language. A first attempt to carry out this step is to perform such a transformation manually, which constitutes a process in which a professional with both UML and Formula skills may be required. It must be noted that such an encoding process may entail a great effort depending on the source CD. 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 have used a MDA–based tool that particularly allows us to automatically carry out the transformation from the CD graphical representation to Formula. More specifically, as described previously, user–defined constraints can be implicit in the model notation or explicitly established by means of OCL constraints. Taking this into account, our plug–in covers the translation of the CD together with the user–defined constraints implicitly represented on it. In Figure 6 we show a snapshot of the plug–in. The idea is that the defined plug–in together with the Formula tool, constitute our overall proposed framework for CD graphical representation to Formula code.

As far as the development of the *CD2Formula* plug–in is concerned, it uses a MDA–based plug–in, which gives support for customizable model–to–text (M2T) transformations. Among the large amount of MDA-based tools in literature, we have chosen the MOFScript Eclipse plug-in [29], which provides support for customizable model–to–text transformations, and which we have used in previous works [30, 31]. As input models, MOFScript can use any model that complies with the EMF [32] metamodel. From these input models, the tool can generate any arbitrary text using a defined set of MOFScript transformations. Each MOFScript transformation consists of transformation rules that 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, as source models of the MOFScript transformations, we use the UML 2.1 metamodel and the specific CD as the model. This CD can be defined using any textual or graphical UML 2 compliant tool that can create models in the XMI format supported by EMF (in particular, we have used the UML2 Eclipse plug-in [33] version 2.1.0, which is based on the UML 2.1.0 specification defined by OMG).

Regarding the generation of the Formula representation of CDs, an important remark must be made. Because of the bounded verification approach followed by Formula, in our proposal we use the `Introduce` Formula instructions in the $CDInstance_{FPM}$ partial model to tell the Formula solver about the user–defined bounds of valid instances we would like for the final solution. The number of instances of class, property, and association in the model should be manually provided by the user as inputs to the plug-in interface by following the guides given in Subsection 4.3. In contrast, the bounds of the `LinkGen`*`ParentChild`* instances will be directly provided by the plug–in according to the bounds of the instances of the specific child.

Since such bounds have to be provided before carrying out the transformation from the CD to its Formula representation, we first need to ask the user for such information, which depends on the specific CD. Taking this into account, we have defined two sets of MOFScript transformations to be executed subsequently. These sets are used (1) to generate a java graphical user interface (GUI), which asks the user for the required number of instances, and (2) to create the Formula representation of the CD (whose $CDInstance_{FPM}$ partial model is generated taking into account the values inserted by the user by the previously generated GUI interface). We want to note that these MOFScript transformations have been defined based on our CD–to–Formula translation proposal, so such transformations are independent of the specific CD and do not have to be modified to translate other CD.

Finally, we have integrated the two sets of MOFScript transformations into the *CD2Formula* Eclipse plug–in, so that the translation from a CD to its Formula representation can be carried out automatically. More specifically, the plug–in provides a menu option "Transformations/UML Class diagram to Formula" (see step labeled 1 in Figure 6) available for each UML CD (specified as `.uml` extension files), which allows the execution of the MOFScript transformations, which (1) dynamically create the GUI interface asking the user for the required information, that is, bounds of instances per class, property and association (see step labeled 2), (2) retrieve the values inserted by the user in the interface, and finally (3) generate the Formula representation of the CD, that is, the `FormulaSpecifications.4ml` file (see step labeled 3), using such values. Finally, the resulting `.4ml` extension file is used by the Formula tool for reasoning about the CD. An Eclipse distribution with the *CD2Formula* plug–in, together with relevant documentation and examples are available from [34]. We encourage the interested reader to try it out.

6.2. Reasoning about the case study

In this section, we briefly present some results and experiences we have obtained from the application of our framework to the case study. Using our proposal, we have been able to not only find conforming object models for the original diagram, but we have also validated interesting business constraints, which have shown the existence of anomalies in the CD, under specific situations.

First, as described previously, we have created the CD model in Figure 1 using the UML2 Eclipse plug–in (as a `.uml` extension file).

Second, taking the resulting file as an input model, we have used the menu option the *CD2Formula* plug-in provides to automatically generate the Formula representation of the CD. In this step, as described previously, we have manually established the bounds of instances we want Formula to generate. In particular, for the example we have chosen low bounds (we have set all bounds to 5 excluding, taking into account the generalization set constraints, the `InstancePerson` data type, which has been set to 15, and `InstanceEmployee` data type, which has been set to 10). The process has resulted in a `.4ml` extension file with more than 450 lines. We want to note from the number of lines that a manual definition of the Formula file would constitute a tedious and delicate work. We have also included the Formula representation of the business OCL constraints labeled from (1) to (4) in Figure 1. Finally, we have begun the reasoning process. In particular, we have carried out several experiments, among which we show, as an example, two of them devoted to: (1) to reason about the CD of Figure 1 to find out if it is satisfiable, that is, if there exists a conforming object model for the CD, and (2) to verify more complex business constraints.

As far as the first experiment is concerned, we have started from the Formula file with the chosen bounds, and we have used the Formula finder to reason about the model, finally getting a positive result. In particular, the tool has returned an instantiation of the model verifying all the established constraints, including the intermediate facts derived from the given facts in the partial models. More specifically, the tool has generated a valid set of instances of the corresponding `Instance`, `Link`, and `Slot` data types, conforming to the Formula model (and thus the original CD), and verifying the Formula constraints (and thus the CD/OCL constraints). Additionally, the tool returns the new entries Formula generates in its fact-base from the defined rules at the *M2* level (for example, of the `supClass`, the `inhsProp`, and the `inhsAsso` rules), taking as a starting point the instances given at level *M1*. In particular, in Figure 7 we show some of the generated Formula instances, which have been distributed into two columns. Additionally, we have slightly compacted the resulting instances in order to make them more legible. More specifically, in lines from 1 to 38 of Figure 7 we show some of the instances we provide to Formula in the domain $CDModel_{FM}$ at level *M1* (see lines from 1 to 15), and some of the new entries that Formula generates for the case study by executing such rules (see lines from 16 to 38), representing the hierarchical structure and the inherited associations. In lines from 40 to 88 of Figure 7, we show some of the instances Formula generates representing the conforming model. As an example, in this figure we can see in bold text the relationships of the full time employee with ID `ifte2` (see line 46), which corresponds simultaneously to *employee* `ie4` (see line 85), and to *person* `ip2` (see line 83). Such an employee is not married but his(her) parent

```
1  classPerson is Class("Person", false)
2  classEmployee is Class("Employee", false)
3  classFullTimeEmployee is
                   Class("FullTimeEmployee", false)
4  classPartialTimeEmployee is
                   Class("PartialTimeEmployee", false)
5  classCompany is Class("Company", false)
6  classDepartment is Class("Department", false)
7 ...
8  family is Association("family",
     classPerson, 0, star, classPerson, 0, 2)
9  Assigned is  Association("Assigned",
                 classDepartment,1,1,classEmployee,3,star)
10  worksIn is Association("worksIn",
                 classCompany,1,star,classEmployee,1,star)
11 makesUpOf is Association("makesUpOf",
                classDepartment, 1, star, classCompany,1,1)
12  genPersonEmployee is
     Generalization(classEmployee,classFullTimeEmployee)
13  genEmployeeFullTimeEmployee is
    Generalization(classEmployee,classPartialTimeEmployee)
14 genEmployeePartialTimeEmployee is
     Generalization(classPerson,classEmployee)
15 ...
16 supClass(classPerson,classEmployee)
17 supClass(classEmployee,classFullTimeEmployee)
18 supClass(classEmployee,classPartialTimeEmployee)
19 supClass(classPerson,classFullTimeEmployee)
20 supClass(classPerson,classPartialTimeEmployee)
21 inhsAsso(classPerson,family)
22 inhsAsso(classPerson,marriage)
23 inhsAsso(classEmployee,Assigned)
24 inhsAsso(classEmployee,worksOn)
25 inhsAsso(classEmployee,worksIn)
26 inhsAsso(classEmployee,family)
27 inhsAsso(classEmployee,marriage)
28 inhsAsso(classFullTimeEmployee,drives)
29 inhsAsso(classFullTimeEmployee,Assigned)
30 inhsAsso(classFullTimeEmployee,worksIn)
31 inhsAsso(classFullTimeEmployee,worksOn)
32 inhsAsso(classFullTimeEmployee,family)
33 inhsAsso(classFullTimeEmployee,marriage)
34 inhsAsso(classPartialTimeEmployee,Assigned)
35 inhsAsso(classPartialTimeEmployee,worksIn)
36 inhsAsso(classPartialTimeEmployee,worksOn)
37 inhsAsso(classPartialTimeEmployee,family)
38 inhsAsso(classPartialTimeEmployee,marriage)
39 ...
40 ip1 is InstancePerson(5,classPerson)
41 ip2 is InstancePerson(7,classPerson)
42 ip3 is InstancePerson(8,classPerson)
43 ip4 is InstancePerson(66,classPerson)
44 ip5 is InstancePerson(67,classPerson)
45 ifte1 is InstanceFullTimeEmployee(82,
                    ClassFullTimeEmployee)
46  ifte2 is InstanceFullTimeEmployee(85,
                     ClassFullTimeEmployee)
47  ipte1 is InstancePartialTimeEmployee(110,
                     ClassPartialTimeEmployee)
48  ipte2 is InstancePartialTimeEmployee(111,
                     ClassPartialTimeEmployee)
49  ipte3 is InstancePartialTimeEmployee(112,
                    ClassPartialTimeEmployee)
50  ipte4 is InstancePartialTimeEmployee(113,
                    ClassPartialTimeEmployee)
51  ipte5 is InstancePartialTimeEmployee(114,
                    ClassPartialTimeEmployee)
52  ie1 is InstanceEmployee(22,classEmployee)
53  ie2 is InstanceEmployee(36,classEmployee)
54  ie3 is InstanceEmployee(39,classEmployee)
55  ie4 is InstanceEmployee(109,classEmployee)
56  ic1 is InstanceCompany(53,classCompany)
57  ic2 is InstanceCompany(108,classCompany)
58  id1 is InstanceDepartment(21,classDepartment)
59  id2 is InstanceDepartment(52,classDepartment)
60  ipr1 is InstanceProject(81,classProject)
61  ipr2 is InstanceProject(84,classProject)
62  Linkfamily(4,family,ip1,ip5)
63  Linkfamily(6,family,ip2,ip3)
64  LinkAssigned (19,Assigned,id2,ie4)
65  LinkAssigned (20,Assigned,id1,ie1)
66  LinkAssigned (23,Assigned,id2,ie3)
67  LinkAssigned (24,Assigned,id1,ie2)
68  LinkworksOn(35,worksOn,ipr2,ie2)
69  LinkworksOn(37,worksOn,ipr1,ie1)
70  LinkworksOn(38,worksOn,ipr2,ie3)
71  LinkworksOn(40,worksOn,ipr1,ie4)
72  LinkmakesUpOf(51,makeUpOf,id2,ic2)
73  LinkmakesUpOf(54,makeUpOf,id1,ic1)
74  Linkmarriage(65,marriage,ip4,ip5)
75  Linkmarriage(68,marriage,ip3,ip4)
76  Linkmarriage(69,marriage,ip5,ip3)
77  Linkdrives(80,drives,ipr2,ifte1)
78  Linkdrives(83,drives,ipr1,ifte2)
79  LinkworksIn(96,worksIn,ic1,ie3)
80  LinkworksIn(107,worksIn,ic2,ie4)
81  ...
82  LinkGenPersonEmployee(-68,
                  genPersonEmployee,ip4,ie2)
83  LinkGenPersonEmployee(-57,
                 genPersonEmployee,ip2,ie4)
84  LinkGenPersonEmployee(-40,
                  genPersonEmployee,ip1,ie3)
85  LinkGenEmployeeFullTimeEmployee (-93,
          genEmployeeFullTimeEmployee ,ie4,ifte2)
86  LinkGenEmployeeFullTimeEmployee (-66,
          genEmployeeFullTimeEmployee ,ie2,ifte1)
87  LinkGenEmployeePartialTimeEmployee (-61,
         genEmployeePartialTimeEmployee ,ie3,ipte4)
88  LinkGenEmployeePartialTimeEmployee (55,
         genEmployeePartialTimeEmployee ,ie1,ipte5)
```

Figure 7: Formula instances generated for the case study.

is the *person* with ID `ip3` (see line 63) (inherited association). In particular, `ifte2` drives (see line 78) and works on (see line 71) the *project* with ID `ipr1` (inherited association). Additionally, the employee works in the *company* with ID `ic2` (see line 80).

Regarding the second experiment, we have validated different business constraints, which one–by–one, have been translated into the Formula language and included into the Formula representation of the original CD, in order to be verified. Included in this kind of experiment, we have considered hypothetical system conditions and later, we have validated specific constraints to know whether such constraints are satisfiable under such system situations.

As an example of such business constraints, let's suppose that we have a company with a specific structure of employees, projects, and departments conforming with the CD of Figure 1, and we want to prove that the team members of a project belong to the department that is officially driving the project. Such a constraint can be defined as:

```
context Project inv:  self.participant->
        forAll(e:Employee|e.department.name=self.driver.name)
```

In order to verify such a constraint, we have slightly modified the Formula file considered in the previous experiment. In particular, we have included on such a file the translation to Formula of the previous OCL constraint. Secondly, we have considered specific Formula instances in the $CDInstance_{FPM}$ partial model at level M0, simulating a specific system composed by one company made up of two departments (identified by `d1` and `d2`), two projects (identified by `p1` and `p2`), and six employees, in such a way that: (1) four employees work on project `p1` and the other two work on project `p2`, and (2) three employees belong to the department `d1` and the other three to the department `d2`. Considering this company structure and the OCL constraint, as it would be expected, Formula proves that such an instantiation of the model is not possible due to the conflict in the multiplicity constraints (in particular, the multiplicities of the associations `Assigned` and `worksOn`) and the defined OCL constraint, caused by insufficient staff. In particular, Formula has labeled the model as unsatisfiable and, analyzing the failed queries, they have given us the clue of that conclusion.

7. Discussion and related work

The formalization and analysis of UML Class Diagrams has motivated a significant number of proposals. As described previously, most of these proposals tackle the verification process by the translation of the model to other languages that preserve its semantics, and the resulting translation is used to reason about the design by checking a predefined set of correctness properties. We evaluate our proposal by comparing it to relevant related work regarding the following dimensions: (1) main tool features, (2) support for UML class diagram elements, (3) support for OCL elements and (4) performance.

Our approach follows a bounded verification strategy which guarantees termination by limiting the search space. This is a popular strategy for the verification of UML class diagrams and it is used in approaches such the one proposed in [35], the approach given in [36] and the one presented in [7, 37], which tackle the verification of UML class diagrams without OCL constraints, and the proposal given in [38, 39, 40, 41], the method presented in [42, 43], in [44], in [45, 46, 47, 48], and in [49], which focus on UML class diagrams with OCL constraints.

From these works, we select for our comparison the approaches currently supported by automated tools. These tools are: *CD2Alloy* [7, 37], *UML2Alloy* [38, 39, 40, 41], *UMLtoCSP* (and its successor *EMFtoCSP*) presented by [42, 43], *MaxUSE* [44], and *USE* with the *SMT-based ModelFinder* plug-in, and the *USE ModelValidator* plug-in, respectively [45, 46, 47, 48]. Next, we present the comparison of our proposal with such tools, leaning on a set of tables in which we have used the following general notation: (i) an empty cell represents that the authors do not mention anything about the aspect in question, (ii) the symbol "N/A" indicates that the aspect or characteristic is not applicable to the specific tool for some reason, and (iii) "No"/"Yes" means that the work explicitly claims that the system *does not*/*does* support the aspect in question. In other cases we have included in the corresponding cell the specific aspect.

Table 13: Tool features

			Tool features											
			General aspects		Problem addressing				Essential characteristics					Usability
Id	Ref	Tool name	Approach type	Presented as	Ability to determine that the problem exists	Ability to identify the cause of the problem	Model Instance Validation	Support for partial state completion	Accepts as input a standard model notation	Ability to verify without requiring any type of additional manual annotation	Automatic verification capability, without user interaction	Provides understandable results for the user	Integrates effortlessly in SDLC	
1	[7, 37]	**CD2Alloy**	SAT	Eclipse plugin	Yes	Yes	Yes	No	No	Yes	Yes	Yes	No	++
2	[38-41]	**UML2Alloy**	SAT	Stand alone	Yes	Yes	Yes	No	Yes	Yes	Yes	Yes	No, only ArgoUML	+
3	[42-43]	**UMLToCSP/ EMFToCSP**	CSP	Eclipse plugin	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	+++
4	[44]	**MaxUSE**	SMT	Stand alone	Yes	Yes	No	No	No	Yes	Yes	Yes BUT not a model instantiation	No	
5	[45-48]	**SMT-based ModelFinder**	SMT	USE plugin	Yes	Yes	Yes	Yes	No	Yes	Yes	Yes	No	++
		USE ModelValidator	SAT	USE plugin	Yes	Yes	Yes	Yes	No	Yes	Yes	Yes	No	++
6		**CD2Formula**	SMT	Eclipse plugin	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	++

7.1. *Tool features*

We summarize the main features of each tool in Table 13, which is organized according to four categories: *General aspects*, which mainly remarks the underlying solver used by each proposal, *Problem addressing*, which refers to different levels of problem reasoning, including also the support for partial state completion, *Essential characteristics*, which refers to a list of characteristics that are regarded as essential for any method for model satisfiability as stated in [50], and finally, *Usability*.

Regarding the *General aspects* category, when comparing the verification technology we find that both *UML2Alloy* and *CD2Alloy* use the *Alloy Analyzer* [51] as their underlying verification tool, which in turn uses a SAT solver internally. Compared to *Alloy*, *Formula* has a more expressive language and employs modern satisfiability modulo theories (SMT) solver, instead of reduction to SAT [13]. The *USE* tool supports two different model search tools: the *SMT-based ModelFinder* plug-in (which supports *Z3* and partially *metaSMT*), and the *USE ModelValidator* plug-in (based on *relational logic/SAT* solver), so that the *USE* framework can be used with both plug-ins. More specifically, *SMT-based ModelFinder* uses the Eclipse Modeling Framework (EMF) as underlying UML/OCL metamodel (so, the USE model under verification has to be transformed from the USE format to EMF). The resulted model is transformed into an instance of the *SMT-LIB* metamodel, that is, a precise SMT problem, which is finally passed to the SMT solver [45, 47]. In the case of *USE ModelValidator*, it uses relation logic, *Kodkod*, *Alloy*, and SAT solvers. The main flow is thereby similar to SMT-based model finding, i. e., the *USE ModelValidator* uses *Kodkod* to transform the model, which itself uses *Alloy* to eventually generate an equivalent SAT formulation to be solved [47, 48]. *MaxUSE* integrates USE with the Z3 SMT Solver. Finally, both *UMLtoCSP* and its successor *EMFtoCSP*, which extends *UMLtoCSP* to deal with EMF metamodels, use Constraint Satisfaction Programming to analyze the models [42, 43], in particular, the ECLiPSe Constraint Programming System (from now on we refer to these tools simply as *EMFtoCSP*).

As for the *Problem addressing* category, most of the presented tools can identify problems in the given input models and they can also be used to inspect and validate correct models. The exception is *MaxUSE*. It is a tool for finding achievable OCL constraints and conflicts for consistent UML class diagrams, also based on the *USE* modelling tool [52]. *MaxUSE* is able to find achievable constraints based on user rankings and constraint conflicts for inconsistent UML class diagrams. Users can rank individual constraints to distinguish their importance, and the tool shows what constraints are achievable and which ones cause the conflicts, finding the different ways of achieving a maximum number of the chosen invariants. However, it is important to note that if the UML class diagram is consistent no instantiation of the model is given. Such instantiation is useful to validate the model constraints (this aspect is considered in the *Essential characteristics* category). Another interesting issue considered in this category is the possibility of supporting the use of partial knowledge to help generate models. Partial state completion is only given by *CD2Formula*, together with *EMFtoCSP* and *USE* with both plug-ins [45]. Similar to our proposal, in all tools (excluding *MaxUSE*, as we will explain later) the designer has to provide some problem bounds in order to derive a decision problem with a finite search space.

Among the *Essential characteristics* desirable for any method for model satisfiability [50], we note whether the tool accepts as input a standard model notation, which constitutes another difference in the tools. Most of the presented tools accept UML class diagrams as input language, except for the three *USE*-based tools (which accept their specific USE notation), and the *CD2Alloy* (that uses the CDs and ODs sublanguages of UML/P, which is a conceptually refined and simplified variant of UML designed for low-level design and implementation [7]). Another characteristic identified in [50], and which is related with the previous one, is whether the proposal integrates seamlessly into the software development life cycle (SDLC). While *CD2Formula* and *EMFtoCSP* are presented as Eclipse plug-ins, and can use CDs defined using any textual or graphical UML 2 compliant tool that can create models in the XMI format supported by EMF, other tools are not so easy to integrate into the SDLC. In the case of *UML2Alloy*, the user can not use a UML-based tool other than ArgoUML [38]. *USE*-based tools are considered as not able, because of the use of *USE* grammar, since the use of this tool implies the conversion of the UML class diagrams to USE specification (we note that there exists a prototype conversion tool from XMI to USE grammar [53], but support for the latest USE versions is not available). In this category, *CD2Formula*, together with *EMFtoCSP* are rated positively in all the considered aspects.

Finally, we have devoted the last column in Table 13 to evaluate the tools' "*usability*", aimed at giving an insight about the verification process performed by each one. To evaluate this aspect, we considered as less usable the tool requiring extra steps. We have excluded *MaxUSE* from this evaluation since its purpose is different than the other tools. In the case of *CD2Alloy*, as described previously, starting from a CD (in the not usual UML/P format), the tool produces, if exists, the corresponding OD (also in UML/P), generating as intermediate step, an *Alloy* module. Since the OD UML/P format is difficult to read and understand, the user can obtain a graphical representation of the resulted OD, but it requires 1) launching the stand alone *Alloy Analyzer* tool [54], 2) loading the *Alloy* module into it, and finally 3) verifying the model. As for the *UML2Alloy* tool, the user has to define the CD in ArgoUML, exporting it in XMI format and giving it as input to the *UML2Alloy* tool. Finally, the tool outputs an `.als` file with the *Alloy* module and, if possible, the corresponding *Alloy* instance as well as a graphical image of the OD. Finally, the *Alloy Analyzer* [54] can be used to validate the model, as with *CD2Alloy*. In the case of the *EMFToCSP* tool, Eclipse can be used as the overall base tool since the source CD in EMF can be created by using the range of Eclipse EMF tools. The tool generates both a `.xmi` and a `png` image of the resulted OD. Regarding the process followed by the two *USE* plug-ins, in addition to the previous remarks made about them, we note that in the particular case of the *USE ModelValidator* tool, previous versions [45] require the specification of the problem bounds to be provided by means of a *configuration* file, that is, the modeler has to edit a text file containing key-value pairs to setup values for certain keys. This process requires a deep understanding of the keys that exist and the syntax to enter the values. To help with this task, the authors present in [48] a configuration GUI for the easy specification of problem bounds.

7.2. Support for UML Class Diagrams

Aspects regarding support for UML Class Diagram elements are presented in Table 14. In particular, *UML2Alloy* translates CD features by mapping each CD construct to a semantically equivalent *Alloy* construct. This fact prevented this proposal from taking full advantage of the expressive power of *Alloy*, which would be necessary to cover the rich features of CDs. More specifically, this approach misses support for several CD features, because they have no direct counterpart in *Alloy* (for example, primitive types different from integers, *multiple inheritance*, or *strong composition*). Another drawback to remark is that, as we have experienced using this tool, version 0.5.2 does not support multiplicity ranges other than the 'trivial' ones (`1`, `0..1`, `0..*`, etc.) and a manual addition of suitable OCL expressions is needed (more details can be seen in [40]). Thus, the weakness of this proposal is the lack of support of UML features often used in CDs representing real system models.

As described previously, *CD2Alloy* uses the CDs and ODs sublanguages

Table 14: Support for UML Class Diagrams

			UML support									
			Data types			Inheritance			Associations			
Id	Ref	Tool name	Primitive/ enum	User-defined	Attributes	single inheritance	multiple-inheritance	generalization sets	binary	n-ary	Association classes	Composition
1	[7, 37]	**CD2Alloy**	Yes	No	Yes, of any multiplicity	Yes	Yes	No	Yes		No	Strong Composition
2	[38-41]	**UML2Alloy**	Only integer, enum	No	Only with multiplicity of 1	Yes	No	Only incomplete and disjoint	Yes	No	No	No, but by refactoring
3	[42-43]	**UMLToCSP/ EMFToCSP**	Not enum		Only of a basic type and of multiplicity of 1	Yes	No	Yes	Yes	Yes	Yes	No, but by refactoring
4	[44]	**MaxUSE**				Yes			Yes			
5	[45-48]	**SMT-based ModelFinder**	boolean, integer, enum (no String)			Yes			Yes	No	No	
		USE ModelValidator	boolean, integer, enum (String only as tokens)			Yes			Yes	Yes	Yes	
6		**CD2Formula**	Integer, String, boolean, enum	Yes	Yes, of any multiplicity	Yes	Yes	Yes	Yes	No	Yes	Strong Composition

of UML/P [7]. The tool uses a deep embedding strategy, which defines new concepts within *Alloy* for some CD constructs, instead of using direct, immediate counterpart constructs in *Alloy* for the translation of CDs features as *UML2Alloy* [38, 39]. In this way, the authors provide support for the representation of more complex CD features when compared to *UML2Alloy*, such as *strong composition* or *multiple inheritance*. Still, *CD2Alloy* does not have support for *association classes* and *user-data types*.

We consider that our approach is more efficient when processing CDs using class inheritance. *CD2Alloy* flattens the inheritance hierarchy creating a list of attributes and associations for each class and all its super classes. This leads to a translation that is more difficult to implement and more computationally complex since the flattening of the inheritance hierarchy requires a global analysis of the CD and, in the worst case, its reconstruction functions may result in a module whose size is quadratic in the size of the input CD. This leads to a larger formula for the SAT solver used by *Alloy* [7]. In contrast, our approach does not need to traverse the class hierarchy structure and produces more compact specifications than *CD2Alloy* when class inheritance is used in a CD.

As for the remained approaches considered in the comparison, we would like to note that all of them provide an implementation proposal for the main

CD elements (class, attributes, associations) differing from the support they provide for other not–so–common elements, being the proposal *EMFtoCSP* [42, 43] one of the most complete ones. In particular, *association classes* are only supported by *EMFtoCSP* and *USE-ModelValidator* [45], *strong composition* is somehow supported by *EMFtoCSP* by refactoring it as an association with additional OCL constraints, while the translation of *generalization sets* is only provided by *EMFtoCSP*. Regarding *n-ary* associations, we note that they are only supported by *EMFtoCSP* and *USE-ModelValidator*. As for the proposal given by *MaxUSE*, we just deduce the translation of the main CD elements (class, attributes, associations), since no explanation about other elements is given in the paper [44].

Although not being a proposal included in our comparison, a special remark must be made regarding the approach given by the authors of Formula in [14, 18]. To our knowledge, our approach together with the one proposed in [14, 18], is the only one that considers a MOF-like metamodeling framework, which turns out to have several advantages. One of the reasons to choose a metamodeling approach is mainly because it allows us a theoretical coverage of the UML language features. We consider that representing just the model pins the result with the specific problem domain, while representing the domain using a metamodeling approach helps to better identify domain-model dependencies and to assess more generic domain models. We also consider that providing a translation that captures the level–based structural distribution can contribute to ease the application and understandability of the representation of a CD/OCL model into Formula. Additionally, our MDA model–to–text transformation process takes advantage of the structure of the M2/M1 Formula representation of CDs, since both the input and the source models used in our MDA process are dispersed into the various parts proposed by MDA (metamodel/model). Aimed at comparing our MOF–like metamodeling proposal with [14, 18], it includes substantial additions. We propose a more faithful representation of the basic UML metamodel and instance domain elements [2]. Furthermore, we also give

Table 15: Support for OCL

			OCL support										
				Collection operators							Other operations		
							Iterators						
Id	Ref	Tool name	Collection types	size	nonEmpty/ isEmpty	union	any	collect	select	forAll/ exists	implies	Built-in properties: oclIsKindOf oclIsTypeOf oclAsType	Closure
1	[7, 37]	**CD2Alloy**	N/A										
2	[38-41]	**UML2Alloy**		Yes	Yes	Yes	No	Yes	Yes	Yes	Yes	oclIsKindOf ✓ oclIsTypeOf ✗	No
3	[42-43]	**UMLToCSP/ EMLToCSP**	sets, bags, sequences, ordered sets	Yes		Yes				Yes	Yes	oclIsKindOf ✗ oclIsTypeOf ✗ oclAsType ✗	No
4	[44]	**MaxUSE**	Not explicitly described by the proposal										
5	[45-48]	**SMT-based ModelFinder**	Set ✓ Bag *partially* orderedSet, Sequence (concept but not implemented)	Yes			Partially (depends on the context)	*Partially*	Yes	Yes	Yes	oclIsTypeOf *Partially* oclAsType *Partially* (depends on the context)	No
		USE ModelValidator	Set ✓ Bag ✗ orderedSet, Sequence (concept but not implemented)	Yes	Yes		Only if the condition limits the number of elements to zero or one (the result must be deterministic)	Yes		Yes	Yes	oclIsTypeOf ✓ oclAsType ✓	Yes
6		**CD2Formula**	Set and Bag	Yes	Yes	Yes	No	Yes	Yes	Yes	Yes	No	Yes

support for the translation of more metamodel elements (such as full support to multiple inheritance, strong composition, and property types other than *Integer*, *String*, and *Boolean*, *user–defined data types* and enumerations, multiplicities of properties, etc.), thus providing a richer framework.

To sum up, it is worth noting that complete support to UML CD elements, such as a wide number of *data types* (including *user-defined types*), *multiple inheritance*, *strong composition* or *association classes*, has not been normally tackled by related works. Regarding the supported CD elements, our proposal could be considered as the most complete one.

7.3. Support for OCL

Regarding the support for OCL (see Table 15), we consider that *UML2Alloy* [38, 39], *USE ModelValidator* [47, 48] and our proposal have the most comprehensive support for OCL. Most significantly, *CD2Alloy* does not have support for OCL constraints. This is an important drawback that has been overcome by our proposal. *EMFtoCSP* has other limitations in the supported OCL fragment, including that `oclIsTypeOf`, `oclIsKindOf` or `oclAsType` cannot be applied to collection expressions, or that *transitive closure* is not supported.

Regarding the three *USE*-based tools, we have to note that, while the *USE ModelValidator* plug-in seems to be more complete than the *SMT-based ModelFinder*, as stated in [45], *USE ModelValidator* and our proposal have several similarities as far as number of supported elements is concerned. For example, while *USE ModelValidator* gives support to `any`, `oclIsTypeOf`, and `oclAsType`, it does not implement OCL `bags` in OCL (in contrast to *CD2Formula*). As for *MaxUSE*, authors in [44] do not mention any aspect regarding the supported OCL fragment. They remark that they plan to exploit multiple SMT solvers for reasoning over a larger number of OCL constraints.

Although not included in our comparison, another similar and interesting work related to the support of OCL constraints is the one given in [22]. In [22] the authors define a fragment of OCL called OCL–lite and prove the encoding of such a fragment in the description logic $\mathcal{ACLI}$, 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 to ours, we have attempted 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 `isEmpty` operator), considering more useful OCL features such as `union` or `collect`. In contrast, OCL–lite supports `oclIsTypeOf`, and `oclAsType` applied to user defined classes.

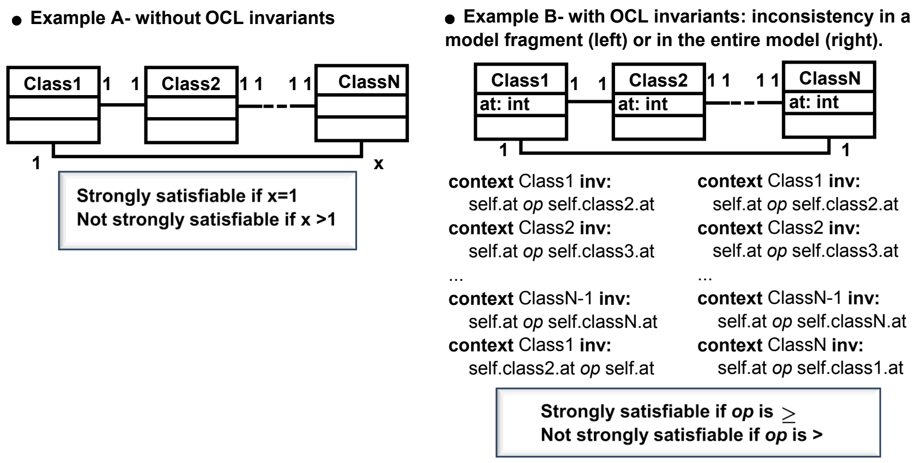
In conclusion, our approach gives support to operators that are not straightforward, such as *transitive closure*, not normally included in related works (in fact, it is not considered by *UML2Alloy*). Nevertheless, a number of OCL elements are not supported by our existing proposal, such as type related operations (`oclIsKindOf, oclIsTypeOf`, and `oclAsType`), some of them not considered by other proposals either.

7.4. Performance

In order to evaluate the performance of our proposal, we have carried out three computational of experiments. The main goal of the first and second experiments is to show the performance and scalability our proposal with different types of CDs (with and without OCL constraints), while the third experiment aims at comparing the performance of our tool with *CD2Alloy*.

7.4.1. Experiment I

In the first experiment, we have used a benchmark suite of CDs offering various experimental CD settings. This benchmark comprises a wide number of satisfiable CDs, which have been designed considering different sizes and complexities. In particular, such CDs have been defined starting from very simple CD examples, which were subsequently increased by including more elements and more complex elements, such as (multiple) inheritance, strong composition, association classes, user data types (including enumerations), reflexive association relationships, etc. We want to note that in [34] the reader can find relevant documentation about the graphical representation, the Formula codification, and the Formula instances generated during the reasoning process of some of the CDs used in our benchmark. The execution time of the tool could be broken down into two steps: the automatic generation of the Formula code file from the CD by the *CD2Formula* plug-in, and (2) the reasoning process itself, carried out by the Formula solver *Z3*. As far as the first step is concerned, once the user inserts the number of valid instances required for the final solution, the automatic generation of the Formula code file takes an insignificant amount of time to compute (ranging from few milliseconds to one or two seconds, depending on the size of the CD). Regarding the second step, as it would be expected, the computation time depends on the size and complexity of the specific CD about which we want to reason. In particular, for each CD in our benchmark, we have carried out several experiments using different values for the parameter `n` in the `Introduce(f,n)` terms of the $CDInstance_{FPM}$ partial model. In this way, for each CD, we have experimented with several configurations, as many as the different values of `n`. One of the lessons learned during the course of our experiments is related to the importance of constraints included in the CD considering not only user-defined constraints, such as multiplicities of proper-


● Example A- without OCL invariants
Class1
Class2
ClassN
1
1
1 1
1 1
1
x
Strongly satisfiable if x=1
Not strongly satisfiable if x >1
● Example B- with OCL invariants: inconsistency in a model fragment (left) or in the entire model (right).
Class1
at: int
Class2
at: int
ClassN
at: int
1
1
1 1
1 1
1
1
Strongly satisfiable if *op* is ≥
Not strongly satisfiable if *op* is >

```
context Class1 inv:
   self.at op self.class2.at
context Class2 inv:
   self.at op self.class3.at
...
context ClassN-1 inv:
   self.at op self.classN.at
context Class1 inv:
   self.class2.at op self.at
```

```
context Class1 inv:
   self.at op self.class2.at
context Class2 inv:
   self.at op self.class3.at
...
context ClassN-1 inv:
   self.at op self.classN.at
context ClassN inv:
   self.at op self.class1.at
```

Figure 8: Examples with and without OCL constraints

ties and associations, but also other business requirements explicitly defined in the model as OCL constraints. Another lesson learned refers to scalability. In particular, our implementation works considerably fast for small CDs, but it does not scale to handle CD properties and associations with high multiplicities and a large number of unknowns.

7.4.2. Experiment II

The second experiment is based on the experiment presented in [42]. The goal is to compare the performance in satisfiable (Sat) and non-satisfiable problems (Unsat). Briefly speaking, we have considered three scenarios. In the first one (presented in Figure 8 as Example A), a CD with multiplicities and no explicit OCL constraints has been considered. We remark that in that figure, if the value of `x` were upper than `1`, the CD would become unsatisfiable, thus, considering different values of `x`, we can evaluate the behavior of our tool both with satisfiable and unsatisfiable versions of the CD. In the second and third scenarios (depicted in Figure 8 as Example B), we present a model with OCL constraints. In these two scenarios we consider all association ends to have a multiplicity of 1..1, making the structural problem satisfiable. Additionally, we define `n` constraints, each defining a relationship between the value of the attribute `at` in a class `i` and the value of the corresponding object in class `i+1`.

Table 16: Execution times for 2, 4, 8, 16 and 32 classes

	Example A (no OCL)		Example B (with OCL invariants)			
			Model fragment (left)		Entire model (right)	
n	**Sat(s)**	**Unsat(s)**	**Sat(s)**	**Unsat(s)**	**Sat(s)**	**Unsat(s)**
2	0,07	0,07	0,12	0,12	0,11	0,12
4	0,13	0,13	0,25	0,27	0,25	0,26
8	0,24	0,25	0,90	1,65	0,90	1,64
16	0,71	0,75	4,74	4,79	5,06	7,19
32	2,78	2,56	47,02	64,56	47,87	74,89

Depending on the chosen operator op ($>$ or $\geq$), the CD may be strongly satisfiable or not. In the second scenario (Example B left) the consistency arises due to the incompatibility of two constraints involving `Class1` and `Class2`. The third scenario (Example B right) considers a case where the incompatibility arises from the interaction of all constraints in the model, which establishes a cyclic dependency on the values of the attributes of all classes. The experiments have been tested on an ordinary computer, Intel(R) $Core^{TM}$ i5 CPU, 3.2 GHz, with 4 GB RAM, running Windows 7 Enterprise, using CDs with different sizes (2, 4, 8, 16 and 32 classes). The resulted execution times are presented in Table 16. We note that the worse-case scenario for our approach would correspond to the unsatisfiability version of the third scenario. From the results it can be inferred that in most cases, as it would be expected, the performance in unsatisfiable situations is worse than the corresponding satisfiable cases, increasing the time as the number of classes become high. It is worth remarking the results obtained from the experiments performed in the model with OCL constraints (Example B), where the execution time suffers from a considerable increment compared with the experiments performed with the version without OCL invariants (Example A). Although if we focus on the execution times in the worst-case scenario (Example B right), they tend to significantly increase as the number of classes gets higher, in average it could be said that the method offers good performance in small and medium-size models, even considering OCL constraints.

7.4.3. Experiment III

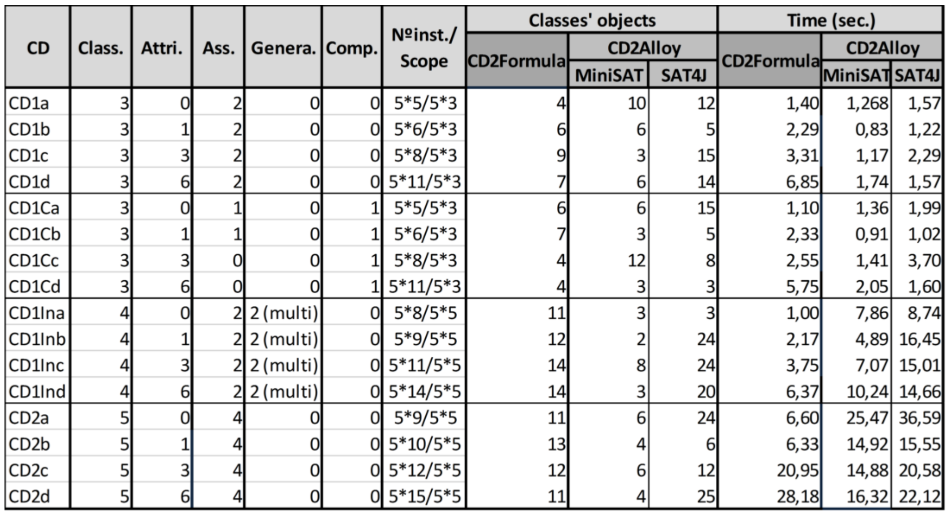
In order to compare the performance of our tool with others, and taking into account that the authors of the *Formula* tool remarked the closeness of both Formula and *Alloy*, in our third experiment we have chosen an *Alloy*-based tool. More specifically, we have chosen *CD2Alloy* [7] instead of *UML2Alloy* [38, 39] because the former tool lacks support for important UML elements, such as *strong composition* or *multiple inheritance* (among other basic CD features). Several remarks are relevant when comparing the tools.

CD2Alloy analysis is based on an exhaustive search for instances of the module, bounded by a user-specified *scope*, which defines the maximal number of objects in the resulting instance OD. In *CD2Formula*, we use the `Introduce(f,n)` generation option, which adds, at most, `n` terms of the form `f` to the partial model. Taking this into account, in our experiments we have considered `n=5` in the case of *CD2Formula*, and a `scope=(number of classes in the CD)*5`, attempting to match as much as possible the reasoning processes with both tools. We note, however, that the use of such values for the `scope` and `n` may be misleading because of the difference in meanings. Finally, since *CD2Alloy* does not display the total time (constructing the formula and solving it) it takes to run the verification, we have taken the *Alloy* module (generated by the tool *CD2Alloy*) and run it into the *Alloy Analyzer* [54] (version `Alloy 4.2` platform independent), which delivers *Alloy*'s output for timings. In particular, we have performed our experiments in *Alloy Analyzer* twice, using two different SAT solvers: `SAT4J`, which corresponds to the default pure Java solver which runs on every platform and operating system, and `MiniSat`, recommended if required faster performance.

Experiments have been done using our *CD2Formula* plug-in and *CD2Alloy* version 1.0.0 available from [37], on the same computer than the previous experiments. We conducted each experimental test three times and report the lower computation time of the three tests in order to avoid possible interferences from the operating system and background processes.

In particular, we have taken a set of CDs from those in our benchmark of

Table 17: Comparison results



CD	Class.	Attri.	Ass.	Genera.	Comp.	Nºinst./ Scope	Classes' objects			Time (sec.)		
							CD2Formula	CD2Alloy		CD2Formula	CD2Alloy	
								MiniSAT	SAT4J		MiniSAT	SAT4J
CD1a	3	0	2	0	0	5*5/5*3	4	10	12	1,40	1,268	1,57
CD1b	3	1	2	0	0	5*6/5*3	6	6	5	2,29	0,83	1,22
CD1c	3	3	2	0	0	5*8/5*3	9	3	15	3,31	1,17	2,29
CD1d	3	6	2	0	0	5*11/5*3	7	6	14	6,85	1,74	1,57
CD1Ca	3	0	1	0	1	5*5/5*3	6	6	15	1,10	1,36	1,99
CD1Cb	3	1	1	0	1	5*6/5*3	7	3	5	2,33	0,91	1,02
CD1Cc	3	3	0	0	1	5*8/5*3	4	12	8	2,55	1,41	3,70
CD1Cd	3	6	0	0	1	5*11/5*3	4	3	3	5,75	2,05	1,60
CD1Ina	4	0	2	2 (multi)	0	5*8/5*5	11	3	3	1,00	7,86	8,74
CD1Inb	4	1	2	2 (multi)	0	5*9/5*5	12	2	24	2,17	4,89	16,45
CD1Inc	4	3	2	2 (multi)	0	5*11/5*5	14	8	24	3,75	7,07	15,01
CD1Ind	4	6	2	2 (multi)	0	5*14/5*5	14	3	20	6,37	10,24	14,66
CD2a	5	0	4	0	0	5*9/5*5	11	6	24	6,60	25,47	36,59
CD2b	5	1	4	0	0	5*10/5*5	13	4	6	6,33	14,92	15,55
CD2c	5	3	4	0	0	5*12/5*5	12	6	12	20,95	14,88	20,58
CD2d	5	6	4	0	0	5*15/5*5	11	4	25	28,18	16,32	22,12

CDs and we have carried out the reasoning process using both *CD2Alloy* and *CD2Formula*. In particular, in Table 17 we present the setup and the performance results from our experiments. More specifically, for each CD, the table shows its main characteristics regarding the number and complexity of its elements: (1) the number of classes, attributes, associations, generalizations, and strong composition elements, (2) the number of instances of classes, attributes, and relationships (associations, generalizations, and strong compositions) we asked the *CD2Formula* tool to generate, which is represented by *N inst*, and the *Scope* value for *CD2Alloy*, (3) the number of classes' objects included in the ODs generated by both tools, distinguishing both SAT solvers, and (4) the total time, in seconds, it took to run the verification in both tools.

We have chosen the set of CDs attending to the following criteria: the set of CDs goes from small CDs with simple CD elements (classes, associations, and properties) to bigger CDs with more complex CD elements (including elements, such as strong composition or multiple inheritance). In particular, we have initially started from two simple CDs (CD1 and CD2), and later we have extended CD1 with strong composition and inheritance elements (CD1C and CD1In). For each variation of CD (CD1, CD1C, CD1In, and CD2), we have carried out different experiments including 0, 1, 3, and 6 attributes (CD♯a, CD♯b, CD♯c, and CD♯d). All the experiments shown in the table correspond to passed verifications (i.e., whether the CD is satisfiable). The CDs included in the table do not define specific business constraints. We have to note that association classes were excluded from these experiments since *CD2Alloy* does not give support for them.

Regarding the results of the experiments, we want to note that in both tools the verification ran quite fast in relatively small CDs, yielding similar results for both tools (being slightly better in *CD2Alloy*, especially when using the `MiniSAT` solver), while solving time increased for bigger CDs. In particular, results shown that although the verification time increased with both tools as the number of classes and associations were bigger, the time required by *CD2Alloy* was greater on average (both with `MiniSAT` and with `SAT4J`). Especially remarkable are the experiments carried out with the CDs including multiple inheritance where, as we have commented previously, since their translation in *CD2Alloy* is more difficult to implement, it required more computational time to solve, in contrast to *CD2Formula*, which needed less than a third, comparing with `MiniSAT`, and a half, comparing with `SAT4J`, of the average. Although obtaining in average better results than with *CD2Alloy*, as presented previously, a weakness of our proposal is related to the increase of associations and, especially, of properties. In this latter case, the Formula solver takes considerably more time to compute because of the constraints it has to handle in order to ensure that the slots are instances of the corresponding property, including in the suitable class/association. In the future, we plan to improve our translation for properties in order to get better scalability of their verification. The reader can find more relevant documentation regarding the conducted experiments on [34].

When comparing *CD2Formula* to related research tools, we consider that our work presents a comprehensive support to class diagrams and OCL constraints as defined in the UML standards that is not matched by other tools. Still, *CD2Formula* lacks support for some modeling constructs supported by other approaches that we have not considered in our study. In any case, we consider that a complete, production-ready model verification tool would be the result of the combination of several existing research approaches, including the ones described in this paper.

8. Conclusion and future work

In this paper we present a framework to reason about UML/OCL models based on the Constraint Logic Programming paradigm. The main contribution of our work is the translation of a UML model into a Constraint Satisfaction Problem following a multilevel Meta–Object Facility (MOF) like framework. Model reasoning can be automated using the model–finding tool Formula. We have also identified a fragment of OCL, which can be checked for finite satisfiability, while being considerably expressive. We also show how to translate such an OCL fragment to Formula by giving, as an intermediate step, a representation of the OCL constraints as FOL expressions. Regarding tool support, we also provide an implementation of our CD–to–Formula proposal as an Eclipse plug–in. It can be used to reason about UML models by checking correctness properties and generating model instances automatically using Formula.

Although our plug–in (1) gives support for the automatic translation to Formula of a CD, including constraints implicit in the model, and (2) provides an approach for the manual translation of OCL constraints explicitly established using OCL, the automatization of this latter aspect constitutes a remaining work. Another interesting issue is related to the selection of suitable verification bounds. As stated by [55, 56], choosing suitable verification bounds constitutes a non-trivial process, as there is a trade-off between the verification time and the confidence in the result. In fact, this aspect has proven itself to be a major limiting factor since existing tools provide little support in this choice. In fact, tools turn to set inadequate default values or force users to manually define these boundaries. The reason why tools provide such little support is mainly because choosing optimal bounds automatically is as complex as the verification problem itself, requiring heuristics or approximate methods [55, 56]. In our particular case, our proposal does not yet tackle the automatic selection of the optimal bounds, but would constitute an interesting issue to be tackled in near future. Finally, we can make a remark regarding the automatic visualization of the resulting instance as an object diagram (OD), characteristic supported by other related tools such as *EMFtoCSP* [42]. In particular, to ease the generation of the ODs, we could implement this step, for example, following a MDA-based approach, so that the corresponding OD could be created automatically in a format conforming with the UML2 Eclipse plug-in. The resulting OD could be taken as input for another available UML modeling tool, which would check the conformance with the source CD.

9. Acknowledgements

This work has been partially supported by the Spanish Ministry of Economy and Competitiveness (project EDU2016-79838-P) and by the Academy of Finland (project VAMOLA). This work has also been partially supported by the Electronic Component Systems for European Leadership Joint Undertaking under grant 520 agreement No 737494. This Joint Undertaking receives support from the European Unions Horizon 2020 research and innovation programme and Sweden, France, Spain, Italy, Finland and Czech Republic.

References

[1] J. Bézivin, Model driven engineering: an emerging technical space, in: Proceedings of GTTSE 05, Springer-Verlag, Berlin, 2006, pp. 36–64.

[2] OMG, UML 2.4.1 Superstructure Specification, august, 2012. Available at: http://www.omg.org/. Last visited on July 2018.

[3] OCL, Version 2.3.1, http://www.omg.org/spec/OCL/2.3.1/PDF. Last visited on July 2018.

[4] A. Calì, D. Calvanese, G. D. Giacomo, M. Lenzerini, A Formal Framework for Reasoning on UML Class Diagrams, in: Proceedings of ISMIS 02, Springer, 2002, pp. 503–513.

[5] J. Cabot, R. Clarisó, D. Riera, Verification of UML/OCL Class Diagrams using Constraint Programming, in: Proceedings of ICSTW 08, IEEE Computer Society, 2008, pp. 73–80.

[6] B. Beckert, U. Keller, P. H. Schmitt, Translating the Object Constraint Language into First-order Predicate Logic, in: Proceedings of FLoC 02, 2002, pp. 113–123.

[7] S. Maoz, J. O. Ringert, B. Rumpe, CD2Alloy: Class Diagrams Analysis Using Alloy Revisited., in: Proceedings of MoDELS 11, 2011, pp. 592–607.

[8] J. M. Bruel, R. B. France, Transforming UML Models to Formal Specifications, in: Proceedings of OOPSLA 98, Springer, 1998, pp. 78–92.

[9] W. E. McUmber, B. H. C. Cheng, A general framework for formalizing UML with formal languages, in: Proceedings of ICSE 01, IEEE Computer Society, 2001, pp. 433–442.

[10] M. Broy, M. V. Cengarle, H. Grönniger, B. Rumpe, Considerations and Rationale for a UML System Model, in: K. Lano (Ed.), UML 2 Semantics and Applications, John Wiley & Sons, Hoboken, 2009, pp. 43–60.

[11] J. Osis, U. Donins, Formalization of the UML Class Diagrams, in: Evaluation of Novel Approaches to Software Engineering, Vol. 69 of Communications in Computer and Information Science, Springer, 2010, pp. 180–192.

[12] FORMULA - Modeling Foundations, http://research.microsoft.com/en-us/projects/formula. Last visited on July 2018.

[13] E. K. Jackson, E. Kang, M. Dahlweid, D. Seifert, T. Santen, Components, platforms and possibilities: towards generic automation for MDA, in: Proceedings of EMSOFT 10), ACM, 2010, pp. 39–48.

[14] E. K. Jackson, T. Levendovszky, D. Balasubramanian, Reasoning about Metamodeling with Formal Specifications and Automatic Proofs, in: Proceedings of MODELS 11, Springer, 2011, pp. 653–667.

[15] B. Pérez, I. Porres, Reasoning About UML/OCL Models Using Constraint Logic Programming and MDA, in: Proc. of ICSEA, 2013, pp. 228–233.

[16] B. Pérez, I. Porres, An Overall Framework for Reasoning About UML/OCL Models Based on Constraint Logic Programming and MDA., International Journal on Advances in Software 7 (1 & 2) (2014) 370–380.

[17] G. Bezhanishvili, L. Moss, Undecidability of First-Order Logic, educational module, for the NSF-sponsored project on Learning Discrete Mathematics and Computer Science via Primary Historical Sources, 26 pp, 2009.

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

[19] G. Booch, J. Rumbaugh, I. Jacobson, Unified Modeling Language User Guide, The (2Nd Edition) (Addison-Wesley Object Technology Series), Addison-Wesley Professional, 2005.

[20] M. Gogolla, M. Richters, Transformation Rules for UML Class Diagrams, in: Proceedings of UML 98, Springer, 1999, pp. 92–106.

[21] A. Kleppe, J. Warmer, S. Cook, Informal Formality? The Object Constraint Language and Its Application in the UML Metamodel., in: Proceedings of UML 98, Springer, 1998, pp. 148–161.

[22] A. Queralt, A. Artale, D. Calvanese, E. Teniente, OCL-Lite: Finite reasoning on UML/OCL conceptual schemas, Data Knowl. Eng. 73 (2012) 1–22.

[23] J. Jaffar, M. J. Maher, K. Marriott, P. J. Stuckey, The Semantics of Constraint Logic Programs, J. Log. Program. 37 (1998) 1–46.

[24] A. Bundy, Tutorial Notes: Reasoning about Logic Programs, in: Proceedings of LPSS 92, Springer, 1992, pp. 252–277.

[25] B. Beckert, R. Hähnle, P. H. Schmitt, Verification of Object-Oriented Software. The KeY Approach, Springer, Berlin, Heidelberg, 2007.

[26] T. Baar, The Definition of Transitive Closure with OCL - Limitations and Applications, Vol. 2890 of LNCS, 2003, pp. 358–365.

[27] A. G. Garis, A. Cunha, D. Riesco, Translating Alloy Specifications to UML Class Diagrams Annotated with OCL, in: Proceedings of SEFM 11, Springer, 2011, pp. 221–236.

[28] J. Cabot, E. Teniente, Transformation techniques for OCL constraints, Sci. Comput. Program. 68 (3) (2007) 179–195.

[29] MOFScript Eclipse plug in, https://marketplace.eclipse.org/content/mofscript-model-transformation-tool. Last visited on July 2018.

[30] B. Pérez, I. Porres, Authoring and Verification of Clinical Guidelines: a Model Driven Approach, J. Biomed. Inform. 43 (4) (2010) 520–536.

[31] E. Domínguez, B. Pérez, M. A. Zapata, Towards a Traceable Clinical Guidelines Application: A Model Driven Approach, Methods of Information in Medicine 46 (6) (2010) 571–580.

[32] EMF Development team, The Eclipse Modeling Framework website: http://www.eclipse.org/modeling/emf/. Last visited on July 2018.

[33] The Eclipse UML2 project, https://www.eclipse.org/modeling/mdt/?project=uml2. Last visited on July 2018.

[34] CD2Formula Eclipse plug—in, http://www.unirioja.es/cu/beperev/CD2FormulaTool.html. Last visited on July 2018.

[35] M. Cadoli, D. Calvanese, G. D. Giacomo, T. Mancini, Finite satisfiability of UML class diagrams by Constraint Programming, in: Proc. of the 2004 International Workshop on Description Logics (DL2004), Vol. 104, 2004.

[36] H. Malgouyres, G. Motet, A UML Model Consistency Verification Approach Based on Meta-modeling Formalization, in: Proceedings of the 2006 ACM Symposium on Applied Computing, SAC '06, ACM, 2006, pp. 1804–1809.

[37] CD2Alloy: Class Diagrams Analysis Using Alloy (1.0.0), http://www.se-rwth.de/materials/cd2alloy/. Last visited on July 2018.

[38] B. Bordbar, K. Anastasakis, UML2ALLOY: A tool for lightweight modelling of discrete event systems, in: Proc. of IADIS AC, 2005, pp. 209–216.

[39] K. Anastasakis, B. Bordbar, G. Georg, I. Ray, UML2Alloy: A Challenging Model Transformation, in: Proc. of MoDELS 07, Vol. 4735 of LNCS, 2007, pp. 436–450.

[40] K. Anastasakis, B. Bordbar, G. Georg, I. Ray, On challenges of model transformation from UML to Alloy, Software & Systems Modeling 9 (1).

[41] UML2Alloy Reference Manual (Version: 0.52), http://www.cs.bham.ac.uk/~bxb/UML2Alloy/files/uml2alloy_manual.pdf. Last visited on July 2018.

[42] J. Cabot, R. Clarisó, D. Riera, On the verification of UML/OCL class diagrams using constraint programming, Journal of Systems and Software 93 (2014) 1–23.

[43] C. A. González Pérez, F. Buettner, R. Clarisó, J. Cabot, EMFtoCSP: A Tool for the Lightweight Verification of EMF Models, in: Proceedings of the First International Workshop on Formal Methods in Software Engineering: Rigorous and Agile Approaches (FormSERA), 2012, pp. 44–50.

[44] H. Wu, MaxUSE: A Tool for Finding Achievable Constraints and Conflicts for Inconsistent UML Class Diagrams, in: 13th International Conference of Integrated Formal Methods (IFM 2017), 2017, pp. 348–356.

[45] N. Przigoda, F. Hilken, J. Peters, R. Wille, M. Gogolla, R. Drechsler, Integrating an SMT-Based ModelFinder into USE, in: Proceedings of the MoDeVVa@MoDELS, 2016, pp. 40–45.

[46] N. Przigoda, R. Wille, R. Drechsler, Ground setting properties for an efficient translation of OCL in SMT-based model finding, in: Proceedings of the ACM/IEEE 19th International Conference on Model Driven Engineering Languages and Systems, 2016, pp. 261–271.

[47] M. Soeken, R. Wille, M. Kuhlmann, M. Gogolla, R. Drechsler, Verifying UML/OCL Models Using Boolean Satisfiability, in: Proc. of the Conference on Design, Automation and Test in Europe, 2010, pp. 1341–1344.

[48] F. Hilken, M. Gogolla, User assistance characteristics of the USE model checking tool, in: Proceedings of the Third Workshop on Formal Integrated Development Environment (F-IDE@FM 2016), 2016, pp. 91–97.

[49] T. Massoni, R. Gheyi, P. Borba, Formal Refactoring for UML Class Diagrams, in: Proceedings of the 19th Brazilian Symposium on Software Engineering (SBES), 2005, pp. 152–167.

[50] J. Cabot, R. Clarisó, UML/OCL Verification in practice, in: First International Workshop on Challenges in Model-Driven Software Engineering, 2008, pp. 25–31.

[51] D. Jackson, Software Abstractions: Logic, language, and Analysis., MIT Press, 2006.

[52] M. Gogolla, F. Bttner, M. Richters, USE: A UML-based specification environment for validating UML and OCL, Science of Computer Programming 69 (1) (2007) 27–34.

[53] W. Sun, E. Song, P. C. Grabow, D. M. Simmonds, XMI2USE: A Tool for Transforming XMI to USE Specifications, in: ER Workshops, Vol. 5833 of Lecture Notes in Computer Science, Springer, 2009, pp. 147–156.

[54] Alloy: a language & tool for relational models. Version 4.2 platform independent, http://alloytools.org/. Last visited on July 2018.

[55] R. Clarisó, Bounded Verification of Software Models: Challenges and Opportunities, IN3 Working Paper Series (2014).

[56] R. Clarisó, C. A. González, J. Cabot, Towards Domain Refinement for UML/OCL Bounded Verification, Vol. 9276 of Lecture Notes in Computer Science, Springer, 2015, pp. 108–114.