Support for Domain Constraints in the Validation of Ontologically Well-Founded Conceptual Models

John Guerson, João Paulo A. Almeida, and Giancarlo Guizzardi

Ontology and Conceptual Modeling Research Group (NEMO),
Federal University of Espírito Santo (UFES), Vitória ES, Brazil
{jguerson,jpalmeida,gguizzardi}@inf.ufes.br

Abstract. In order to increase the accuracy of conceptual models, graphical languages such as UML are often enriched with textual constraint languages such as the Object Constraint Language (OCL). This enables modelers to benefit from the simplicity of diagrammatic languages while retaining the expressiveness required for producing accurate models. In this paper, we discuss how OCL is used to enrich a conceptual model assessment tool based on an ontologically well-founded profile of the Unified Modeling Language (UML) that assumes multiple and dynamic classification (called OntoUML). In the approach, OCL expressions are transformed into Alloy statements enabling model validation and assertion verification with the Alloy Analyzer. The tool we have developed allows modelers with no Alloy expertise to express constraints in OCL enriching OntoUML models.

Keywords: Conceptual Model Validation · Domain Constraints · OCL · Alloy · OntoUML

1 Introduction

Conceptual modeling is "the activity of formally describing some aspects of the physical and social world around us for purposes of understanding and communication" [16]. A conceptual model, in this sense, is a means to represent what modelers perceive in some portion of the physical and social world with purpose of supporting the understanding (learning), problems solving and communication, in other words, a means to represent the modeler's conceptualization [11] of a domain of interest.

For a number of years now, there has been a growing interest in the use of Foundational Ontologies (i.e., ontological theories in the philosophical sense) for supporting the activity of Conceptual Modeling giving rise to an area known as Ontology-Driven Conceptual Modeling. In this setting, the OntoUML language has been designed to comply with the ontological distinctions and axiomatic theories put forth by a theoretically well-grounded Foundational Ontology [11]. This language has been successfully employed in a number of industrial projects in several different domains such as Petroleum and Gas, News Information Management, E-Government, Telecom, among others.

OntoUML was designed to address a number of deficiencies in UML from a conceptual modeling standpoint. OntoUML addresses a number of problems in UML regarding ontological expressivity, i.e., the ability of a language to make explicit ontological distinctions present in a domain [11]. These distinctions are important to ensure that the modeler may express as accurately as possible [10] a domain conceptualization, making conceptual models more useful to support the understanding, agreement and perhaps, the construction of information systems.

Despite the advances in the quality of conceptual modeling languages, assessing whether a conceptual model indeed reflects the modeler's intended conceptualization remains a challenging task. In order to support the validation of conceptual models in OntoUML, *Benevides et al.* [2] and *Braga et al.* [3] defined a translation from OntoUML conceptual models to Alloy [12]. The idea is to use the Alloy Analyzer to automatically generate logically valid instances for the OntoUML model at hand. By confronting the user with a visualization of these possible model instances, we are able to identify a possible gap between the set of possible model instances (implied by the model) and the set of intended model instances (which the modeler intended to capture). In other words, in this approach, one can detect cases of instances which conform to the OntoUML model but which do not reflect the modeler's intended conceptualization (due to under-constraining) as well as cases of intended possible instances which are not shown as valid ones (due to over-constraining).

Up to this point, the validation of OntoUML models in this approach has been limited to the formulae implied from its diagrammatic notation. However, in complex domains, there are typically a number of domain constraints which cannot be directly expressed by the diagrammatic notation of the language, but which are of great importance for capturing as accurately as possible the modeler's intended domain conceptualization. In order to address this issue, in this paper, we propose the use of OCL expressions as a mean to enhance the expressivity of OntoUML conceptual models with respect to the explicit representation of domain constraints.

This paper extends the approaches of *Benevides et al.* [2] and *Braga et al.* [3] by defining a translation from OCL to Alloy in compliance with the existing transformation of OntoUML. The OCL subset considered is then determined by the expressivity and significance to the OntoUML modeling language. One of the key differences of OntoUML (and as a consequence of our approach) is that it has a full support for dynamic and multiple classification. Although dynamic and multiple classification are in principle supported by UML class diagrams, most approaches that establish formal semantics and analysis/simulation for these diagrams do not address these features[1]. This renders these approaches less suitable to enable the expression of important conceptual structures that rely on dynamic classification (e.g., the classification of persons into life phases: child, teenager, adult; the classification of persons into roles in particular contexts) as well as multiple classification (e.g., the classification of persons according to orthogonal classification schemes such as: living-deceased, male-female).

[1] Probably, due to the strict correspondence that is often established (even if implicitly) between conceptual modeling languages and programming languages that lack such features.

We define a fully-automated translation from OntoUML+OCL models and implement and incorporate into an OntoUML modeling environment. The tool we have developed allows modelers with no Alloy expertise to write constraints in OCL enriching OntoUML models. In addition to instantiating model instances for model simulation, the tool supports the formal verification of assertions written in OCL. Our overall objective is to support the assessment of conceptual models, retaining the simplicity of a diagrammatic language while coping with the expressiveness required to produce accurate conceptual models.

This paper is organized as follows. Section 2 presents a running example based on a traffic accident ontology in OntoUML and OCL. Section 3 presents the existent transformation from OntoUML to Alloy. Section 4 and 5 describes and applies our OCL translation to validate the running example. Section 6 discusses related work and Section 7 presents some concluding remarks.

2 A conceptual model in OntoUML

This example was inspired by a governmental project that we conducted for a national regulatory agency for land transportation in Brazil. In this domain, travelers are persons taking part of a travel in a vehicle, possibly becoming involved in traffic accidents; traffic accidents involve victims, crashed vehicles and a roadway; and, accidents may involve a number of fatal victims. A particular sort of accident called rear-end collisions is also identified (accidents wherein a vehicle crashes into the vehicle in front of it). **Fig. 1** depicts an OntoUML conceptual model for that domain.

OntoUML extends UML by introducing metaclasses that correspond to ontological distinctions put forth by the Unified Foundational Ontology (UFO). For instance, a class stereotyped as a *kind* provides a principle of application and a principle of identity for its instances [11]. It represents a *rigid* concept, i.e., a class that applies necessarily to its instances (e.g., a Person cannot cease to be a Person without ceasing to exist). A kind can be described in a taxonomic structure where its subtypes are also rigid types known as subkinds (e.g., Man and Woman).

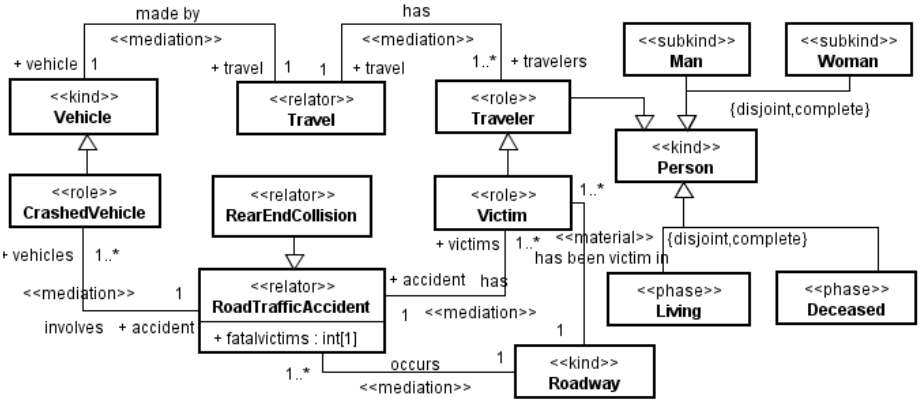

Fig. 1. OntoUML conceptual model of road traffic accidents

A *role*, in turn, is an *anti-rigid* concept, applying contingently to its instances (e.g., a Person can cease to be a Traveller and still exist) and does not provide a principle of identity, instead, it inheres the identity from the unique kind it specializes. A role is also *relational dependent*, i.e., it defines contingent properties exhibited by an instance of a kind in the scope of a relationship (e.g., John plays the role of Victim in an accident contingently and in relation to or, in the context of, that accident).

A *phase* is an anti-rigid concept that it is defined by a partition of a kind and whose contingent instantiation condition is related to intrinsic changes of an instance of that kind (e.g. if Living and Deceased constitutes a Person's phase partition then every Person x is either alive or deceased, but not both. Moreover, a Living Person is a Person who has the intrinsic property of being alive) [11].

A *relator* (e.g. entities with the power of connecting other entities) is a rigid concept and *existentially depends* on the instances it connects through *mediation* relations (e.g., an Accident only exists if Crashed Vehicles, Victims and a Roadway also exist). From an ontological point of view, relators are the truthmakers of the so-called *material* relations. For instance, it is the existence of a particular RoadTrafficAccident connecting Victim X, Crashed Vehicle Y and Roadway Z that makes true the relation has-been-victim-in-roadway(X,Z). In OntoUML, material relations such as this one are considered to be logico-linguistic construction reducible to an analysis of relators, relata and their tying mediation relations. Some UML features such as Bags only occur in an OntoUML model at this linguistic level, i.e., in the context of derived material relations. Other purely linguistic features of UML are dispensed altogether in OntoUML. These include the notion of interface but also the ordered UML meta-attribute (and, consequently, the Ordered Sets or Sequences collections types).

Fig. 2 presents a possible instantiation (simulation) of our model of traffic accidents. It shows a state (current world) where a person (Object1) is classified as Woman and Living and the other person (Object4) as Man and Living, characterizing an example of multiple classification in OntoUML. Both persons, the man and the woman, play the role of travellers in a travel made by the vehicle Object5, crashed in an accident. The simulation shows some situations that may contradict the intended conceptualization for this domain, e.g., that the accident has 4 fatal victims although only one non-fatal victim is identified (Object1) and that the man is a traveller of the vehicle involved in the accident (Object5) but is not a victim of that accident.

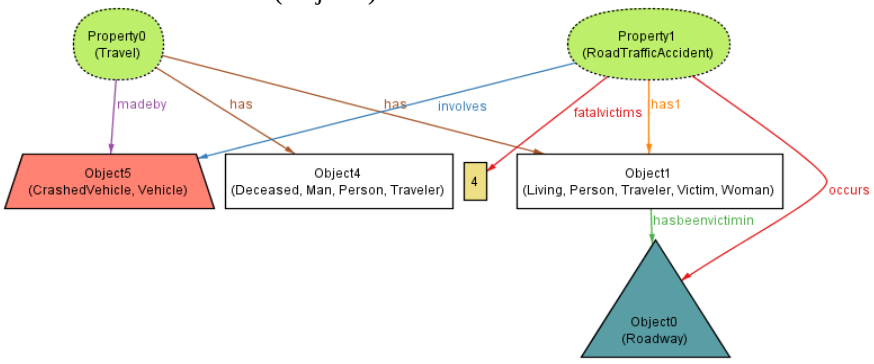

Fig. 2. Automatically generated instantiation without constraints

Note that this simulation result was generated automatically by the Alloy Analyzer (using the existing approach derived from the work of *Benevides et al.* [2] and *Braga et al.* [3]), and thus, using that approach, we are unable to prevent the inadmissible situations from occurring. That will be addressed in our approach by adding domain constraints to the OntoUML model, which will be expressed in OCL and transformed into the Alloy language.

OCL [18] is a (*semi*) formal [5, p.60] language adopted as a standard by the OMG to represent constraints on MOF-based models. It is declarative, textual and based on first-order logic. OCL is used for a variety of purposes such as to express class invariants, attributes and association-end point derivations, query operations definitions and pre- and post-conditions over operations. The subset of OCL considered here includes both invariants and derivations. An invariant is a condition applied to a class in the model. The condition must be true for every class' instance at any point in time, in OntoUML terms, at every world instance, whether past, current, counterfactual or future world [2]. A derivation, in turn, expresses how an association end-point or an attribute can be inferred from other elements of the model. As a structural conceptual modeling language, OntoUML does not target the representation of operations, thus, as a consequence, we do not support the definition of operations and pre- and post-conditions. It is important to emphasize that no change is required in OCL so that it can be used with OntoUML; a subset of OCL can be meaningfully employed to a lightweight extension of UML.

Fig. 3 shows three OCL constraints for the traffic accident domain. The first invariant states that every rear-end collision must involve exactly two crashed vehicles; the second constraint (a derivation) specifies that the attribute *fatalvictims* is derived from the number of deceased victims in an accident; and the third and last constraint (an invariant) states that (i) every traveler of a vehicle participating in an accident is a victim in that accident, and that (ii) every vehicle, in which there is a victim of an accident, is involved in that accident.

```
1  context RearEndCollision inv: self.vehicles->size() = 2
2
3  context RoadTrafficAccident::fatalvictims: int
4  derive: self.victims->select(p | p.oclIsKindOf(Deceased))->size()
5
6  context RoadTrafficAccident inv: self.vehicles->forAll(v |
7  self.victims.oclAsType(Traveler)->includesAll(v.travel.travelers))
8  and self.victims->forAll(p |
9  self.vehicles->includes(p.travel.vehicle.oclAsType(CrashedVehicle)))
```

Fig. 3. OCL domain constrains for the road traffic accident conceptual model

3 From OntoUML models to Alloy specifications

The approach proposed by *Benevides et al.* [2] and *Braga et al.* [3] to support the validation of OntoUML conceptual models uses a lightweight formal method of vali-

dation by defining a (semantic preserving) transformation from OntoUML models to Alloy. The resulting Alloy specification is fed into the Alloy Analyzer tool to generate and visually confront the modeler with possible instances of the model. The set of atoms displayed represent instances of the classes of the OntoUML model and the set of relations between those atoms represent instances of the OntoUML relationships.

Alloy [12] is a declarative and first order logic-based language to describe structures accompanied by a tool to explore and display them graphically. An Alloy specification defines the possible structures of atoms and relations. It is comprised mainly of: signatures with fields and constraints (facts, assertions and predicates). An Alloy signature introduces a set of atoms with relations between them declared as fields of signatures [12, p.35]. An Alloy *fact* is a constraint that must always be respected by the structure of atoms and relations. An Alloy *assertion* is a target for verification, i.e., a boolean expression that the Alloy Analyzer will try to invalidate by examining structures allowed by the specification [12, p.93, 119]. The Alloy Analyzer will either conclude that the assertion is invalid, showing a counterexample for it (a structure that invalidates it), or conclude that it holds for structures up to a certain size (the *scope* of verification). An Alloy *predicate* is a boolean expression that can be used in different contexts, e.g., within facts or within commands for verification and simulation.

Fig. 4 shows part of the Alloy code generated by the translation of the OntoUML model of **Fig. 1**. In line 5, the Alloy signature *Object* represents existentially *independent* entities (e.g. instances of kinds, roles, phases, subkinds). In line 6, existentially *dependent* entities (objectified properties, e.g., relators) are represented by the signature *Property*. In line 7, the abstract signature World represents the states of objects and reified properties. This is required to support the notion of modality that underlies OntoUML and thereby model the dynamics of creation, classification, association and destruction of instances. In each World, Objects and Properties may exist, which is specified using the *exists* field (line 8). Worlds are classified into four sub-signatures: *CurrentWorld*, *PastWorld*, *FutureWorld* and *CounterfactualWorld*. These sub-signatures are specified in a separated module imported as an Alloy library to the specification [2] (line 2). In line 9, the *kind* Person is transformed into a binary relation between the World and the object (instance of person) that exists in that World. The rigidity property of persons is represented by a predicate declaration within a fact statement, as showed in line 15. The rigidity predicate is part of a separated module (imported as a library in line 3), which is committed to specify several ontological properties of OntoUML. Similarly, in line 10, the *role* Victim is transformed into a binary relation between the World and the object existing in that World. In line 11, the *relator* RoadTrafficAccident is transformed into a binary relation between the World and the corresponding objectified property that exists in that World. All the classes in OntoUML follow this transformation to Alloy, i.e., they are Alloy binary relations from worlds to extensions, which allows us to capture dynamic classification. Furthermore, in line 12, the attribute *fatalvictims* is represented as a ternary relation (a triple) between the World, the owner of the attribute and its type (in this case Int). In line 13, the mediation *has* is represented as a ternary relation between the World, the Accident, and the Victim. In addition, the existential dependency between the Accident and its Victims is represented by another imported ontological property

(a predicate) enforcing the immutability of victims in that accident (line 16). Finally, in line 18 and 19, the association end-point *victim* is represented as an Alloy function which receives as parameters the traffic accident, from which the association end-point is reached, and the world instance in which it exists, returning the set of victims related to that accident.

```
1   …
2   open world_structure[World]
3   open ontological_properties[World]
4   …
5   sig Object {}
6   sig Property {}
7   abstract sig World {
8       exists: set Object + Property,
9       Person: set exists:>Object,
10      Victim: set exists:>Object,
11      RoadTrafficAccident: set exists:>Property,
12      fatalvictims: set RoadTrafficAccident set -> one Int,
13      has1: set RoadTrafficAccident one -> some Victim,
14  … }{ … }
15  fact { rigidity[Person,Object,exists] }
16  fact { immutable_target[RoadTrafficAccident,has1] }
17  fun victims[x: World.RoadTrafficAccident, w: World] : set World.Victim
18  { x.(w.has) }
19  …
```

Fig. 4. Resulting Alloy specification from OntoUML

4 From OCL constraints to Alloy constraints

In this section, we define the translation of OCL constraints in Alloy. We assume the transformation of OntoUML to Alloy discussed in the previous section. We use the symbol [[ ]] to denote a function that receives OCL concrete syntax and returns Alloy textual code.

4.1 Invariants and derivations

OCL invariants and derivations are represented in Alloy as facts with formulae which hold in every possible World for all instances of the Context class, as shown in **Table 1**. The body of an invariant is directly transformed into the body of the corresponding Alloy fact. Derivations in turn force the values of attributes and association ends to match the derivation expression. Note that the Alloy counterpart of OntoUML classes, attributes and associations are World fields, referred in the mappings by the expression *w.[[Class]]*, *w.[[attribute]]* and *w.[[association]]*, respectively. The association end-points are represented as functions which receive as parameters the source

object from which the association end-point is reached and the world instance in which it exists. They are referred in the mappings using the Alloy function syntax: *self.[[assocEnd]][w]*, where *assocEnd* represents a function name corresponding to an association end and *self* and *w* are function parameters.

Table 1. Translation of OCL invariants and derivations

OCL constraint	**Alloy statement**
context *Class* inv: *OclExpression*	fact invariant1 { all w: World | all self: w.[[*Class*]] | [[*OclExpression*]] }
context Class::*attribute*:*Type* derive: *OclExpression*	fact derive1 { all w: World | all self: w.[[*Class*]] | self.(w.[[*attribute*]]) = [[*OclExpression*]] }
context Class::*assocEnd*:Set(*Type*) derive: *OclExpression*	fact derive2 { all w: World | all self: w.[[*Class*]] | self.[[*assocEnd*]][w] = [[*OclExpression*]] }

4.2 Expressions

OCL expressions are divided into: if-then-else expressions, let-in expressions, navigational expressions (using the "dot notation") and operation call expressions. The former two can be directly represented in Alloy by equivalent expressions whilst the last one is not considered here since operations are not meaningful in OntoUML. Navigational expressions deserve special treatment as there are different mappings for attribute access and association end navigation. In **Table 2**, we define the mappings for OCL expressions. We use *be* to represent a boolean expression, *expr* to represent an OCL expression, *battr* to represent a boolean attribute, *var* to represent a variable. The dot notation is equivalent in both OCL and Alloy, thus the only difference in the attribute mappings stem from the fact that an OntoUML boolean attribute is represented as an Alloy subset, therefore, the OCL dot operation in this case is mapped to the Alloy operator *in* (the same mapping choice as taken in [6]).

Table 2. Translation of OCL expressions

OCL expression	**Alloy expression**
if *be* then *be1* else *be2* endif	[[*be*]] implies [[*be1*]] else [[*be2*]]
let *var*: Type = *expr* in *be*	let *var* = [[*expr*]] | [[*be*]]
expr.attribute	[[*expr*]].(w.[[*attribute*]])
expr.assocEnd	[[*expr*]].[[*assocEnd*]][w]
expr.battr	[[*expr*]] in (w.[[*battr*]])

4.3 Iterators

Table 3 shows the mappings from OCL iterators into Alloy. The word *col* represents OCL expressions that result in collections and the letter *v* represents variables.

Table 3. Translation of OCL iterators

OCL iterator	Alloy expression
col->forAll(v1,..,vn | *be*)	all v1,..,vn: [[*col*]] | [[*be*]]
col->exists(v1,..,vn | *be*)	some v1,..,vn: [[*col*]] | [[*be*]]
col->select(v | *be*)	{ v: [[*col*]] | [[*be*]] }
col->reject(v | *be*)	{ v: [[*col*]] | not [[*be*]] }
col->one(v | *be*)	#{ v: [[*col*]] | [[*be*]] } = 1
col->collect(v | *expr*)	univ.{ v: [[*col*]], res: [[*expr*]] | no none}
col->isUnique(v | *expr*)	all disj v, v': [[*col*]] | [[*expr*]](v)!=[[*expr*]](v')
col->any(v | *be*)	{ v: [[*expr*]] | [[*be*]] }
col->closure(v| *expr*)	[[*col*]].^{v: univ, res: [[*expr*]] | no none}

OCL iterators are represented in Alloy as quantified formulae and comprehension sets. The *forAll* and *exists* iterators are represented as Alloy formulae quantified universally (keyword *all)* and existentially (keyword *some).* The *select* and *reject* iterators are represented as Alloy comprehension sets (denoted by curly brackets) whilst the *one* iterator is also represented as an comprehension set but using operators such as # (cardinality operator) and = (equality operator) to state that the resulting set must be equal to 1. The *collect* iterator is represented combining comprehension sets, the keyword *univ*, the dot notation and a logical *true* Alloy primitive value (expressed in terms of the keywords *no none)*. The *isUnique* iterator is represented as an Alloy formula universally quantified plus the disjointness keyword *disj*. The *any* iterator is represented by an Alloy comprehension set but with a restriction of usage: the modeler must ensure that the boolean expression evaluates to true in exactly one element of the source collection (the same mapping as in [13]). Finally, the *closure* iterator is represented combining comprehension sets, the transitive closure operator (^) and the Alloy true primitive value, similar to the *collect* mapping previously presented.

4.4 Sets

Alloy supports all the OCL set operations since it is a set-based language. Therefore, the OCL set operations represented in Alloy are: *size*, *isEmpty*, *notEmpty*, *includes*, *excludes*, *includesAll*, *excludesAll*, *union*, *intersection*, *including*, *excluding*, *difference*, *symmetricDifference*, *asSet*, *product*, *sum* and *flatten*. We omit here these mappings since they are rather straight-forward given Alloy's native support for sets.

4.5 Primitive types

Alloy natively supports only the integer and boolean primitive types. They are directly represented in Alloy as well as their operations. However, Alloy does not natively support the OCL *xor* boolean operator and the OCL integer operations *max*, *min* and *abs*. Their mappings to Alloy are shown in **Table 4.** . The supported OCL boolean operations are: *and*, *or*, *implies*, *not* and *xor*; whilst the supported OCL integer operations are the comparison operations (i.e., <, >, <=, >=) as well as some arithmetic

ones (i.e., + *(*sum*)*, - *(*subtraction*)*, * (multiplication), *div*, *floor*, *round*, *max*, *min* and *abs*). Bit width for integers is by default 7 (and thus range by default from -63 to 64).

Table 4. Translation of OCL primitive types

OCL operation	Alloy expression
e1 xor *e2*	([[*e1*]] || [[*e2*]]) && !([[*e1*]] &&[[*e2*]])
e1.max(*e2*)	int[[[*e1*]]] >= int[[[*e2*]]] => [[*e1*]] else [[*e2*]]
e1.min(*e2*)	int[[[*e1*]]] <= int[[[*e2*]]] => [[*e1*]] else [[*e2*]]
e1.abs()	Int[[[*e1*]]] < 0 => [[*e1*]].negate else [[*e1*]]

4.6 Objects operations and meta-operations

Table 5 depicts the object and meta-operations of OCL translated in Alloy where *T* is a type (i.e., a class) in the model. The *oclIsTypeOf* operation means that the object is of the type T but not a sub-type of T. The *oclIsKindOf* operation in turn checks the same condition but including the subtypes of T. The latter is represented by the Alloy subset operator (*in*) whilst the former is represented by an expression combining the operators *in*, *and*, *#* (cardinality*)*, *&* (intersection), + (union) and = (equality), verifying if the object is contained in the set T but not in the union of all subtypes of T (referred by the Alloy expression [[subT1]]+…+ [[subTn]]). The *oclAsType* and *allInstances* operations are directly represented by their source parameter since Alloy is a set-based language.

Table 5. Translation of OCL object operations and meta-operations

OCL operation	Alloy expression
obj.oclIsKindOf(*T*)	[[*obj*]] in w.[[*T*]]
obj.oclIsTypeOf(*T*)	[[*obj*]] in w.[[*T*]] and # w.[[*T*]] & (w.[[*SubT1*]] +…+ w.[[*SubTn*]]) = 0
obj.oclAsType(*T*)	[[*obj*]]
obj.oclIsUndefined()	# [[*obj*]] = 0
Class.allInstances()	w.[[*Class*]]

5 Revisiting the running example

In this section, we revisit the running example, now enriched with domain constraints. We use the OCL transformation discussed in the previous sections and generate valid instances of the traffic accident conceptual model. We further exemplify the use of OCL invariants as assertions subject to verification.

5.1 Simulation

Fig. 5 depicts the code generated by applying our OCL transformation. The generated code is added into the specification resulting from the transformation of the OntoUML model that which was partially presented in **Fig. 4**. All elaborated OCL do-

main constraints are transformed into Alloy facts and thus all instantiations of the OntoUML model will conform to these constraints. The final specification resulting from both OntoUML and OCL mappings is fed into the Alloy Analyzer to generate/check sample structures of the OntoUML+OCL model.

```
context RearEndCollision inv: self.vehicles->size() = 2
fact invariant1 { all w: World | all self: w.RearEndCollision |
# self.vehicles[w] = 2 }
```

```
context RoadTrafficAccident::fatalvictims: int
derive: self.victims->select(p | p.oclIsKindOf(Deceased))->size()
fact derive1 { all w: World | all self: w.RoadTrafficAccident |
self.(w.fatalvictims) = # { p: self.victims[w] | p in w.Deceased } }
```

```
context RoadTrafficAccident inv: self.vehicles->forAll(v |
self.victims.oclAsType(Traveler)->includesAll(v.travel.travelers))
and self.victims->forAll(p |
self.vehicles->includes(p.travel.vehicle.oclAsType(CrashedVehicle)))
fact invariant2 { all w: World | all self: w.RoadTrafficAccident |
(all v: self.vehicles[w] | v.travel[w].travelers[w] in univ.{temp1:
self.victims[w], res: temp1 | no none }) && (all p: self.victims[w] |
p.travel1[w].vehicle[w] in self.vehicles[w]) }
```

Fig. 5. Alloy code resulting from our OCL translation

Fig. 6 depicts a possible instantiation of the traffic accident model enrich with its domain constraints. The figure depicts a current world (a point in time) where a road traffic accident (a rear end collision), between two crashed vehicles resulted in the death of both travelers of the vehicles, and where the two fatal victims were both male persons. All specified OCL constraints are respected (in every point in time, whether past, current or future world). Differently from the unconstrained model that was shown in **Fig. 2**, the derived number of fatal victims is correct, the traveler of the crashed vehicle is indeed a victim of that accident and the *rear end collision* involves two crashed vehicles as required in the definition of this type of accident.

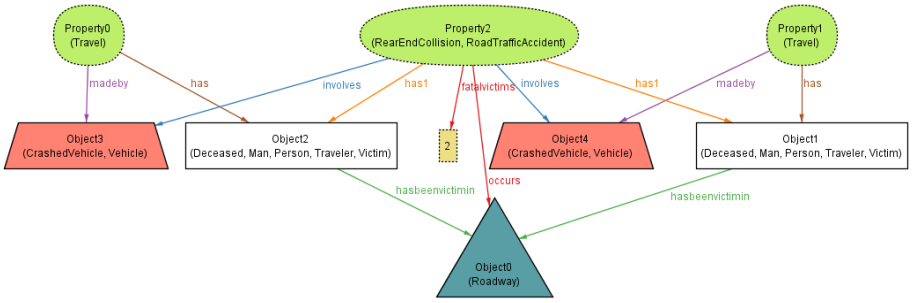

Fig. 6. Automatically generated instantiation with constraints

5.2 Assertion Checking

The same transformation used to generate facts corresponding to OCL invariants can be used to generate assertions which are subject to verification by the Alloy Analyzer. The analyzer will either conclude that the assertion is invalid, showing a counterexample for it (an instance of the model that invalidates the assertion), or conclude that it holds for structures up to a certain size (the *scope* of verification). The approach to verification in Alloy is based on the "small scope hypothesis" which states that, if an assertion is invalid, then it probably has a small counterexample [12, p. 143]. This ensures tractability of assertion verification.

Fig. 7 depicts an assertion written in OCL and its mapping to Alloy. The OCL assertion states that, in a travel, not all travelers are deceased. This is transformed into an assertion plus a check command defining the default scope and the default Alloy bitwidth. Furthermore, in the check command, we also define the number of atoms of the signature World (particularly 1 to ensure a single World atom in the checking).

```
context Travel inv: not self.travelers->forAll(t| t.oclIsKindOf(Deceased))
assert invariant3 {all w: World | all self: w.Travel |
! (all t: self.travelers[w] | t in w.Deceased) }
check invariant3 for 10 but 1 World, 7 Int
```

Fig. 7. OCL assertion and the mapping to Alloy

Fig. 8 shows the counterexample found by executing the check command with the Analyzer, showing thus that the enriched OntoUML model does not guarantee the satisfaction of the assertion. The figure depicts a current world where all travelers of a travel made by a vehicle are actually deceased. The label *$self* means that this particular atom is descendant from the variable *self* in the OCL assertion, where *self* is an instance of Travel. If the modeler intended this assertion to hold (i.e., if he/she believe that the situation is inadmissible in the domain), the OCL expression in the assertion can be considered a fact (an invariant enriching the model), thus preventing this situation from occurring.

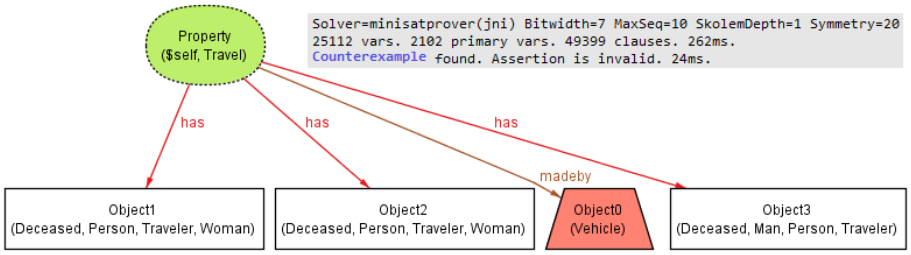

Fig. 8. Counterexample found. Assertion does not hold.

6 Related work

There have been several approaches in the literature to the analysis and validation of UML models and OCL constraints e.g. HOL-OCL [4], USE [9], CD2Alloy [14], UML2Alloy [1]. In particular, a number of these approaches [1], [6], [13], [14], [15] have used Alloy as a lightweight formal method for validating models in UML/OCL. In [1], *Anastasakis et al.* present one of the first extensive approaches for automatic translation of UML+OCL models into Alloy for purposes of model verification and validation. Their tool is called UML2Alloy and although it considers both UML and OCL, it does not support several OCL operators and just a subset of UML is considered. *Cunha et al.* [6] extended the mappings of *Anastasakis et al.* to support, among others, UML qualified associations and dynamics of properties such as the UML *read-only* feature (mutability of properties). They defined a state local signature called Time in the Alloy resulting specification to correctly handle dynamics of properties and pre- and post- conditions. *Kuhlmann et al.* [13] defined a translation from UML and OCL to relational logic and a backwards translation from relational instances to UML model instances (relational logic is the source for the Kodkod SAT-based model instance finder used by Alloy). *Massoni et al.* [15] proposed a transformation of a small subset of UML (class diagrams with classes, attributes and association) annotated with OCL invariants to Alloy. However, they specify the translation only in a systematic and manual way; they do not implement it. Finally, *Maoz et al.* [14] translated UML, particularly class diagrams, to Alloy and then from Alloy's instances back to object diagrams, considering both multiple inheritance and interface implementation. They use a deeper embedding strategy as not all UML concepts are directly translated to a semantically equivalent Alloy construct (for instance, the multiple inheritance feature is transformed to a combination of facts, predicates and functions in Alloy). In addition, they are able to support the analysis of class diagrams, for example, checking if one class diagram is a refinement of some other class diagram [14, p.2]. The translation is fully implemented in a prototype plugin in Eclipse called CD2Alloy, which can (optionally) hide the Alloy resulting specification from the modeler. This translation however does not consider OCL. Besides, the Alloy resulting specification is more difficult to read, less understandable and computationally more complex than other approaches. None of these approaches completely support dynamic and multiple classification, which is essential for ontology-driven conceptual modeling. In fact, besides dynamic and multiple classification, the meta-properties that characterize many of the ontological categories and relations in an ontologically well-founded language are modal in nature. As discussed in [11], the modal distinctions among object types and part-whole relations are paramount from an ontological perspective and play a fundamental role in ontology engineering and semantic interoperability efforts. These modal features (and all language constructs affected by them) require a special treatment in the mapping to Alloy [2] [3]. Our translation of OCL is in pace with all these features.

7 Concluding remarks

In this paper we have presented an approach to validate OCL-enhanced OntoUML models using a lightweight formal method that uses Alloy for model visual simulation and model checking. We have extended the previous work of *Benevides et al.* [2] and *Braga et al.* [3] by defining a translation from OCL constraints into Alloy statements in accordance with the existent transformation of OntoUML. This allows modelers with no Alloy expertise to write constraints in OCL enriching OntoUML models. This work contributes to facilitating the definition of high-quality conceptual models that, albeit grounded on sound ontological distinctions, lacked several domain constraints and did not cover precisely [10] the modeler's intended conceptualization. The approach supports visual simulation of model instances that conform to the enriched OntoUML model as well as supports checking of assertions through model checking.

The translation to Alloy discussed here is fully implemented and incorporated into the OntoUML Lightweight Editor (OLED[2]) developed in our research group. The Alloy code fragments presented in **Fig. 4** and **Fig. 5** are indeed part of the specification as generated by OLED. OLED is an experimental tool for the OntoUML conceptual modeling language that provides instance simulation (via Alloy and its Analyzer) along with other features such as syntax verification, model editing, model verbalization and model transformations (e.g., to languages such as OWL). OLED manipulates OntoUML models using an OntoUML Eclipse metamodel [7][17]. We have employed the Eclipse MDT OCL [8] plugin for OCL syntax verification, auto-complete, parsing and to implement the OCL mappings to Alloy using the visitor pattern. The infrastructure for OCL manipulation and binding to OntoUML is currently being used in order to implement a transformation of OCL to SWRL, building up on an OntoUML to OWL transformation, and enabling the use of OCL constraints for (runtime) inference.

Acknowledgments. This research is funded by the Brazilian Research Agencies CNPq (grants number 310634/2011-3, 311578/2011-0, and 485368/2013-7) and FAPES (grant number 59971509/12).

References

1. Anastasakis, K., Bordbar, B., Georg, G., Ray, I.: On challenges of model transformation from UML to Alloy. Softw. Syst. Model. 9, 1, 69–86 (2010)
2. Benevides, A.B., Guizzardi, G., Braga, B.F.B., Almeida, J.P.A.: Validating modal aspects of OntoUML conceptual models using automatically generated visual world structures. J. Univers. Comput. Sci. 16, 2904–2933 (2011)
3. Braga, B. F. B., Almeida, J. P. A., Guizzardi, G., Benevides, A. B.: Transforming OntoUML into Alloy: towards conceptual model validation using a lightweight formal method. Innov. Syst. Softw. Eng. 6, 1-2, 55–63 (2010)

[2] https://code.google.com/p/ontouml-lightweight-editor/

4. Brucker, A.D., Wolff, B.: HOL-OCL: a formal proof environment for UML/OCL. In: Fiadeiro, J.L. and Inverardi, P. (eds.) 11th International Conference on Fundamental Approaches to Software Engineering, FASE 2008. pp. 97–100 Springer Berlin Heidelberg (2008)
5. Cabot, J., Gogolla, M.: Object Constraint Language (OCL): A Definitive Guide. In: Bernardo, M., Cortellessa V., Pierantonio A. (eds.) 12th International School on Formal Methods for the Design of Computer, Communication, and Software Systems, SFM 2012. pp. 58–90 Springer Berlin Heidelberg (2012)
6. Cunha A., Garis A., Riesco D.: Translating between Alloy specifications and UML class diagrams annotated with OCL. Softw. Syst. Model. (2013)
7. Eclipse EMF, http://www.eclipse.org/modeling/emf/
8. Eclipse MDT OCL, http://www.eclipse.org/modeling/mdt/
9. Gogolla, M., Bohling, J., Richters, M.: Validating UML and OCL models in USE by automatic snapshot generation. Softw. Syst. Model. 4, 4, 386–398 (2005)
10. Guarino, N.: Toward Formal Evaluation of Ontology Quality. IEEE Intell. Syst. 19, 4, 78–79 (2004)
11. Guizzardi, G.: Ontological Foundations for Structural Conceptual Models. Telematica Instituut, The Netherlands (2005)
12. Jackson, D.: Software Abstractions-Logic, Language, and Analysis, Revised Edition. The MIT Press (2012)
13. Kuhlmann, M., Gogolla, M.: From UML and OCL to Relational Logic and Back. In: France, R.B. et al. (eds.) 15th International Conference, MODELS 2012. pp. 415–431 Springer Berlin Heidelberg (2012)
14. Maoz, S., Ringert, J., Rumpe, B.: CD2Alloy: Class Diagrams Analysis Using Alloy Revisited. In: Whittle, J. et al. (eds.) 14th International Conference, MODELS 2011. pp. 592–607 Springer Berlin Heidelberg (2011)
15. Massoni, T., Gheyi, R., Borba, P.: Formal Refactoring for UML Class Diagrams. 19th Brazilian Symposium on Software Engineering (SBES). pp. 152–167 (2005)
16. Mylopoulos, J.: Conceptual Modeling, Databases, and CASE: An Integrated View of Information Systems Development; chapter Conceptual Modeling and Telos; Wiley, Chichester (1992)
17. NEMO OntoUML Infrastructure, http://nemo.inf.ufes.br/en/ontoumlsupport
18. OMG: Object Constraint Language, version 2.3.1 (2012)