ACM/IEEE 18th International Conference on
Model Driven Engineering Languages and Systems

September 27 – October 2, 2015 • Ottawa (Canada)

OCL 2015 – 15th International Workshop on OCL and Textual Modeling: Tools and Textual Model Transformations

Workshop Proceedings

Achim D. Brucker, Marina Egea, Martin Gogolla, and Frédéric Tuong (Eds.)

Published on Oct 2015 v1.0

Workshop URL: http://ocl2015.lri.fr

Editors' addresses:

Achim D. Brucker
SAP SE (Germany)
achim.brucker@sap.com

Marina Egea
Indra Sistemas S.A. (Spain)
msegeo@indra.es

Martin Gogolla
University of Bremen (Germany)
gogolla@informatik.uni-bremen.de

Frédéric Tuong
Univ. Paris-Sud – IRT SystemX – LRI (France)
frederic.tuong@{u-psud, irt-systemx, lri}.fr

Workshop Chairs

Achim D. Brucker	SAP SE (Germany)
Marina Egea	Indra Sistemas S.A. (Spain)
Martin Gogolla	University of Bremen (Germany)
Frédéric Tuong	LRI, Univ. Paris-Sud, CNRS, Université Paris-Saclay (France) — IRT SystemX (France)

Program Committee

Mira Balaban	Ben-Gurion University of the Negev (Israel)
Tricia Balfe	Nomos Software (Ireland)
Achim D. Brucker	SAP SE (Germany)
Fabian Büttner	Inria (France)
Jordi Cabot	ICREA (Spain)
Dan Chiorean	Babeș-Bolyai University (Romania)
Robert Clariso	Universitat Oberta de Catalunya (Spain)
Tony Clark	Middlesex University (UK)
Manuel Clavel	IMDEA Software Institute (Spain)
Carolina Dania	IMDEA Software Institute (Spain)
Birgit Demuth	Technische Universität Dresden (Germany)
Marina Egea	Indra Sistemas S.A. (Spain)
Geri Georg	Colorado State University (USA)
Martin Gogolla	University of Bremen (Germany)
Shahar Maoz	Tel Aviv University (Israel)
István Ráth	Budapest University of Technology and Economics (Hungary)
Bernhard Rumpe	RWTH Aachen (Germany)
Frédéric Tuong	LRI, Univ. Paris-Sud, CNRS, Université Paris-Saclay (France) — IRT SystemX (France)
Claas Wilke	Technische Universität Dresden (Germany)
Edward Willink	Willink Transformations Ltd. (UK)
Burkhart Wolff	LRI, Univ. Paris-Sud, CNRS, Université Paris-Saclay (France)
Steffen Zschaler	King's College (UK)

Additional Reviewers

Javier Luis Cánovas Izquierdo
Klaus Müller

Table of Contents

Preface

Many modeling languages, such as the Unified Modeling Language (UML), advocate the use of graphical notations for modeling. This kind of representations usually allow a nice high level description of the system, while getting into details is more often left to textual modeling or formal languages.

The OCL'15 workshop, as previous editions, was aimed to serve as a forum were researchers and practitioners could exchange ideas, experiences and results for the benefit of both the software engineering community and the standards specifications. The workshop met its goal judging by the versatility of the contributions selected for discussion. Indeed, the reader will find papers dealing with the applicability and limitations of constraints languages for adding precision to modeling and transformation languages, or to bind modeling elements. Also, other works are concerned with the tool support for constraints and query languages, e.g., OCL lazy evaluation for large models, or version control for textual modeling languages. Last but not least, semantics issues bring the question of how specification standards should be enhanced, or how textual and visual notations should be synchronized in order to assure consistency.

Every accepted paper was reviewed by at least three members of the program committee. We wish to thank all authors and participants for their contributions to the workshop, and reviewers for ensuring the proceedings quality. We would like to thank the committees of Models 2015 for making this workshop possible, and organizing the successful surrounding events in Ottawa.

October 2015

Achim D. Brucker
Marina Egea
Martin Gogolla
Frédéric Tuong

Opportunities and Challenges for Deep Constraint Languages

Colin Atkinson[1], Ralph Gerbig[1] and Thomas Kühne[2]

[1] University of Mannheim
{atkinson, gerbig}@informatik.uni-mannheim.de
[2] Victoria University of Wellington
thomas.kühne@ecs.vuw.ac.nz

Abstract. Structural models are often augmented with additional well-formedness constraints to rule out unwanted configurations of instances. These constraints are usually written in dedicated constraint languages specifically tailored to the conceptual framework of the host modeling language, the most well-known example being the OCL constraint language for the UML. Many multi-level modeling languages, however, have no such associated constraint language. Simply adopting the OCL for such multi-level languages is not a complete strategy, though, as the OCL was designed to support the UML's two-level class/instance dichotomy, i.e., it can only define constraints which restrict the properties of the immediate instances of classes, but not beyond. The OCL would consequently not be able to support the definition of deep constraints that target remote or even multiple classification levels. In fact, no existing constraint language can address the full range of concerns that may occur in deep modeling using the Orthogonal Classification Architecture (OCA) as an infrastructure. In this paper we consider what these concerns might be and discuss the syntactical and pragmatic issues involved in providing full support for them in deep modeling environments.

Keywords: OCL; well-formedness; deep modeling; OCA

1 Introduction

Although structural modeling languages such as UML class diagrams can sometimes be used on their own for simple tasks, for more precise modeling they usually need to be supported by accompanying constraint languages. The most well-known language for this purpose is OCL, but today quite a number of other constraint languages are available for use with the UML and other structural modeling languages. Examples include AspectOCL [1], CdmCL [2], EVL [12] and MOCQL [16].

The OCL is commonly used to augment class diagrams with further well-formedness constraints. Constraints can be applied to other UML diagram types but these are not addressed in this paper. The basic role of constraints is to add additional requirements that (token-)models at the instance level must obey in order to be considered instances of their (type-)model at the class level. Since the sets of valid token models are sometimes referred to as the semantics of the (structural) type model, constraints can be regarded as improving the precision of the semantics of the type model.

Since mainstream constraint languages in use today were designed to support today's mainstream structural modeling languages and tools, they are based on the same underlying "two-level" modeling paradigm. In particular, this means that the domain entities considered by the users are regarded as occupying one of two levels – a type level or an instance level. This goes back to the idea that all concepts are either universals or individuals but not both at the same time. As a result, today's constraint languages are often designed in the context of a number of assumptions:

1. constraints are attached to types and are evaluated for instances of these types.
2. constraints are implicitly universally-quantified, i.e., use a "*forAll*"-semantics that applies them all instances of the type they are associated with.
3. there is only one instance level to control, i.e., deep control beyond more than one metalevel boundary is not considered.

These assumptions are not an optimal fit for deep modeling. Constraint languages that are designed to complement deep models, i.e., models with an instantiation depth higher than one, therefore have to be based on a different set of assumptions with implications on what information is necessary to fully describe a constraint in a multi-level context. Designing an approach for constraints that incorporates the extended challenges of deep modeling in an optimal way is a non-trivial task, and requires notational and pragmatic trade-offs that are influenced by many factors. The goal of this paper is not to resolve these trade-offs, but to identify the opportunities and challenges present in the expression of deep constraints and therefore the design of Deep Constraint Languages (DCLs).

As well as supporting well-formedness constraints on structural models, there are a variety of other applications for constraint languages or extensions thereof. These range from queries to transformations, however, for space reasons, in this paper we focus on well-formedness constraints. Most of the ideas apply to similar languages as well, though, for example for deep transformations [5].

Summarising, the goal of this paper is to reevaluate the role of constraints for deep modeling, identify different kinds of constraints, and establish terminology for referring to them. The rest of the paper is structured as follows: In the next section we give a brief overview of deep modeling and the principles it is based on as well as discuss the main implementation choices that can be used to implement deep models. Then, in Section 3 we introduce the different kinds of constraints that can be defined given the two orthogonal classification dimensions that underpin deep modeling. Section 4 then does the same given the multiple ontological levels that can be defined in a deep model. Finally, section 5 concludes with some closing remarks.

2 Deep Modeling

Deep modeling is built around the Orthogonal Classification Architecture (OCA) which provides an alternative way of organizing models compared to the traditional linear modeling stack that underpins mainstream modeling technologies such as UML and EMF. Figure 1 below shows the most widely used variant of the OCA which has the linguistic dimension occupied by two linguistic levels – the modeling language definition in L_1, the ontological model content in L_0 and the real world (W). Although variants with more linguistic levels are conceivable, in this paper we assume the OCA only considers these levels. It can be observed that the ontological (O-levels) and linguistic (L-levels) classification dimensions are orthogonal to each other. This alignment of levels gives the name to the orthogonal classification architecture.

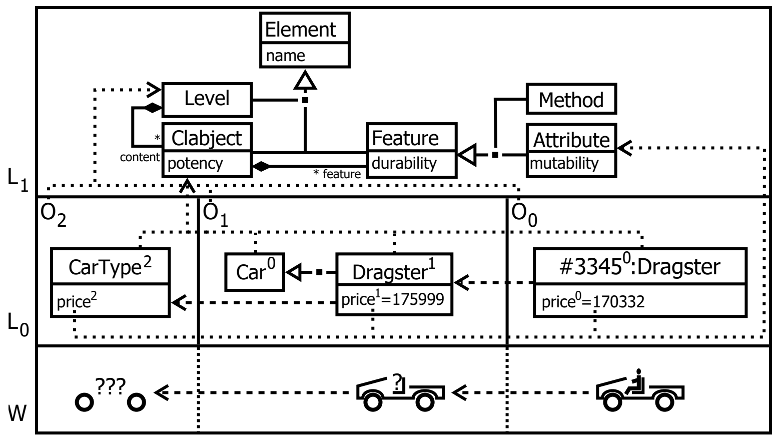


Fig. 1. Three level OCA.

In Figure 1 the linguisistic dimension contains Level L_1, which is often referred to as the "linguistic type model" or (less precisely) "metamodel". This linguistic level defines the basic concepts of *Level*, *Clabject* and *potency* etc. Level L_0 contains the ontological content defined by the end user. In this level the clabjects are organized into multiple ontological classification relationships depending on whether they model objects, types, types of types, etc. in the domain of interest. Liniguistic classification is indicated through dotted vertical classification arrows while ontological classification is indicated through dashed horizontal classification arrows. O_0 contains pure objects (i.e. clabjects with no type facet), O_1 contains normal types (i.e. clabjects that have both a type and an instance facet) and O_2, in this example, contains types (of types) at the top ontological level (i.e. clabjects which have no instance facet).

A second core idea underpinning deep modeling is that classes and objects, which are modeled separately in traditional modeling approaches, are integrated into a single, unified concept known as "clabject". In general, clabjects are classes and objects at the same time, and thus simultaneously possess both type and instance facets, e.g., consider *Dragster* at the middle level, O_1, which is an instance of *CarType* and type for *#3345* at the same time. The third core idea underpinning deep modeling is the notion of deep instantiation which uses the concept of potency, attached as a superscript to the name of each clabject and their features as seen in Figure 1. This concept limits the depth of the instantiation tree of a clabject thus defining the degree to which a clabject can be regarded as a type. In the example, *CarType* with potency two can be instantiated on the following two levels. It is then instantiated with *Dragster* which can be instantiated one level further below and *#3345*, an instance of *Dragster* which cannot be instantiated further as indicated by its potency zero value.

The features of deep modeling that present the biggest opportunities and challenges from the perspective of defining constraints on deep models beyond what is required for traditional "two-level" modeling are:

1. the two distinct, orthogonal classification dimensions,
2. dual facets of model elements (i.e. the existence of linguistic and ontological attributes), and
3. an unbounded number of ontological classification levels.

2.1 Realization Strategies

Modeling environments, including deep modeling environments, are typically built using traditional "two level" technology. It is possible to support multiple, logical modeling levels on top of such two-level physical architectures in two ways. The first is by supporting transformations between chains of two-level models, each capturing a different window on the underlying multi-level model (referred to as the "cascading" style in [6]). The second is by mapping the linguistic metamodel(L_2) to the type level of the implementation platform, and the domain content (L_1) to the instance level of the implementation platform, with some of the relationships in the latter level being regarded as classification relationships. For example, to support the second approach on a Java platform, the elements of the linguistic metamodel (e.g. level, clabject etc.) would be mapped to Java classes, and the L_1 level content would be represented as instances in the JVM. While many commercial tools use the first approach, the second approach is used in the majority of academic tools and ultimately provides the simplest and most flexible way of implementing deep modeling environments. We assume the latter approach in the remainder of this paper, therefore.

In general, approach (b) can be applied in one of two ways. The simplest way is for a modeler to simply model the linguistic metamodel as a class diagram in some existing, "host" modeling environment and then to apply the tenets of deep modeling itself at the instance level using certain well-known patterns [8]. If the host environment has a constraint language (e.g. OCL) this can be used to express limited kinds of constraints over the deep model at the instance level. This approach is taken by Gogolla et al. [8], for example, who represents various deep modeling scenarios within the two-level USE tool (i.e. by modeling L_1 as a class diagram and L_0 as an object diagram) and uses standard OCL to express constraints over the L_0 content. We refer to a constraint language used in such a way as a Standard Constraint Language (SCL).

In contrast, a DCL is a constraint language which includes extra support (explicitly implemented as an extension to the host environment) for the concepts embodied by deep modeling. In other words, a DCL provides additional features for expressing constraints on deep modeling content which are not available in an SCL. As with all DSLs, this support can take the form of additional library functionality (cf. [8]), in which case it is an internal DSL, or it can be provided along with additional syntax, in which case it is an external DSL) [7].

In the following sections we investigate, in turn, the consequences and opportunities resulting from the key features of deep modeling identified previously. In each case we will identify different kinds of constraints that may occur in deep models, show examples of these constraint kinds on a small running example and discuss possible syntactic alternatives for expressing the constraints. We show examples in a suggested syntax which includes notational ideas from three existing constraint languages — the OCL, which plays the role of an SCL in this context, MetaDepth [13] which is an external DSL built on top of the Epsilon Object Language [11], and Deep OCL [10] from Melanee [3], which is an external DSL built on top of the Eclipse MDT OCL.

3 Linguistic, Ontological and Hybrid Constraints

The first important characteristic of deep modeling is that there are two distinct dimensions across which constraints can operate – the linguistic and the ontological dimensions. In principle, modelers may wish to reference the ontological and/or the linguistic dimension when defining constraints. This gives rise to three kinds of constraints – constraints only referencing the ontological dimension, constraints only referencing the linguistic dimension, and hybrid constraints.

3.1 Linguistic Constraints

Linguistic constraints reference concepts in terms of linguistic types and are therefore independent of any ontological types. An example application for them is the definition of the classification semantics of deep modeling. Constraint 1 shows how a linguistic constraint on Figure 2 can be used to define the basic rules of deep instantiation – namely that the potency of an instance of a clabject must be one lower than the potency of that clabject. Constraint 1 is a standard OCL constraint, just applied within the linguistic dimension.

Constraint 1 *The value of the potency of every clabject must be one lower than that of its direct type*

context *Clabject*
getDirectTypes() implies forAll(t | t.potency = potency+1)

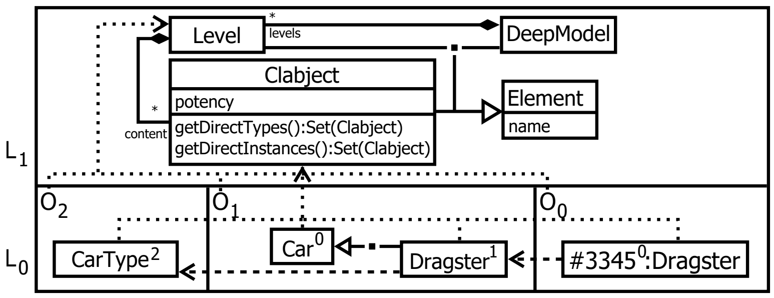


Fig. 2. Linguistic constraints example.

Constraints which enforce certain modeling styles are also possible. For example it is possible to require that clabject potency values must always match level values (as in [13]) or to limit the number of levels available in a deep model (Constraint 3).

Constraint 2 *The value of the potency and level of every clabject must be equal*

> ***context*** *Clabject*
> *self.potency = self.level*

Constraint 3 *The number of levels in a deep model must be 3*

> ***context*** *DeepModel*
> *self.levels implies size = 3*

These constraints are useful in scenarios where the number of ontological levels has to be fixed, e.g. when model execution software is written against a certain level of a deep-model. Like Constraint 1, both Constraint 2 and Constraint 3 are standard OCL constraints whose context is the linguistic meta-model.

3.2 Ontological Constraints

Ontological constraints operate within level L_0 to express well-formedness conditions on the content of ontological levels. These well-formedness conditions include the kind of constraints that end users (i.e. modellers) typically write in conventional modeling environments to constrain the properties of domain instances based on their domain types.

An example of a traditional constraint in the context of Figure 3 is the constraint for a second-hand dealer that the default used *price* of a *ProductType* (e.g., *Lorry*) has to be lower than the respective recommended retail price (*RRP*). Using an OCL-like syntax, a DCL should allow this constraint to be expressed in a way similar to Constraint 4. This constraint is defined "on" *ProductType* and ensures that all ontological instances of *ProductType*, here *Lorry* and *Dragster*, have an *RRP* that is higher than their *price*.

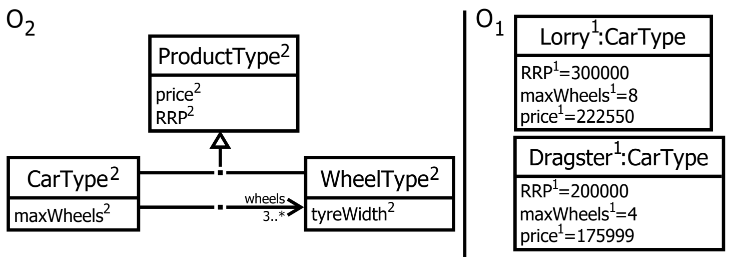


Fig. 3. Ontological constraints example.

Constraint 4 *The value of a ProductType's price attribute has to be smaller than or equal to its RRP attribute*

origin*(1) ProductType*
self.price <= self.RRP

An important notational difference in Constraint 4 is that we use the term "origin" rather than "context" to specify to which element the constraint is attached in order to avoid the traditional meaning associated with the term "context" in OCL. In OCL, the class to which a constraint is attached does not coincide with the evaluation context for the constraint, as OCL implicitly assumes universal quantification of the constraint over all instances of the class. We believe, in order to support more flexibility, it is worthwhile not always making this assumption of implicit universal quantification over instances at the level below. Therefore, in the remainder of the paper we express constraints in the following form —

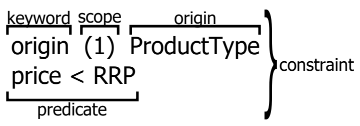


Fig. 4. General structure of a constraint.

where the clabject appearing in the top line after the keyword "origin" is regarded as the definitional anchor for the constraint, whereas the "scope" of the constraint (i.e., to what levels it applies to) is specified by the given range (e.g. *(1)*) as further explained below. The constraint body, defined underneath the header, specifies the predicate that must evaluate to true for all elements of type "origin" within the scope.

3.3 Hybrid Constraints

The constraints discussed in the previous section either operated purely across the linguistic dimension or purely across the ontological dimension. In some situations, however, there is a need to mix the linguistic and ontological dimensions. Such a scenario is shown in Constraint 5 on Figure 5 which ensures that each concrete car must have a price greater than zero and that each car type must have a default price greater than zero. This is achieved using a hybrid constraint whose origin is *CarType*. This constraint uses the linguistic dimension to apply the constraint to all *CarType* instances at levels 1 and 0, and the ontological dimension to ensure that the price of these model elements is greater than zero. Note that Constraint 5 could be expressed more concisely using our proposed scoping mechanism (cf. Section 4.2) and hence demonstrates the latter's utility; Constraints 8 and 9 are more natural examples of hybrid constraints.

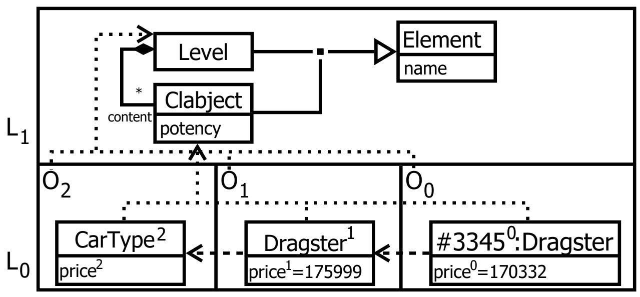


Fig. 5. Hybrid constraints example.

Constraint 5 *The (default) prices for instances of CarType and their instances must be greater than zero*

***origin**(1..2) CarType*
(a) self.level < 2 implies self.price > 0
(b) self._l_.level < 2 implies self._o_.price > 0
(c) self._l_.level < 2 implies self.price > 0
(d) self.^level < 2 implies self.price > 0

The example suggests four different possible notations to define the hybrid constraint. The first notation (a) does not make a syntactic distinction between the dimension in which a called attribute, method, etc. is located. For this approach to work in general, however, it is necessary to ensure that all names used in the L_1 (meta)-model do not appear in L_0 (i.e. in any ontological levels). However, this could be difficult in practice because the L_1 model naturally contains names that could appear in many domains. For example, "level" may very well not just occur in the linguistic dimension to represent the level a model-element resides in, but may also be used to express the location of an elevator in the ontological dimension. To avoid such naming clashes, either names used in the L_1 model must be changed to highly unnatural ones (e.g. “linguistic-residence-level”) etc. or the use of the most natural names (e.g. level) has to be prohibited in domain models. Neither of these would be particularly desirable.

An alternative approach shown in (b) is to introduce a special syntax for selecting between linguistic and ontological features, so that no ambiguity exists even when names from the L_1 (meta)-model are used in an ontological model. For every attribute, method, etc. this approach requires the origin of each referenced element to be identified (i.e. “_l_” for linguistic and “_o_” for ontological), which creates a big overhead when creating hybrid constraints. To minimize this overhead, the notation can be further refined as suggested in (c). This notation assumes the dimension in which the origin of the constraint resides as the default. A dimension must then only be explicitly specified if a user intends to reference the other dimension. In the example the default dimension is the ontological dimension as *CarType* is an O_2-level element.

The notation shown in (d), presented in [14], is used by MetaDepth to resolve linguistic and ontological name disambiguities. MetaDepth allows constraints accessing the ontological and linguistic dimension to be defined in the style of (a). If an ambiguity occurs, the linguistic dimension is marked by prefixing it with a ˆ symbol as shown in (d).

4 Deep Constraints

The aspect of deep modeling which creates the most interesting opportunities is also the most challenging for DCLs: It is the unbounded number of ontological levels that may appear in a deep model. Deep constraints are constraints that may target levels more than one level below and may even have multiple levels in their scope. In the OCA implementation assumed in this paper, linguistic constraints cannot be deep because there are only two linguistic levels. In general, however, deep linguistic constraints may be useful to constrain instances of languages in a language family or ensure consistency across multiple language levels.

Deep Constraints can be classified as either level-specific or level-spanning constraints. In order to clarify the difference it is necessary to introduce some further terminology. More specifically, it is necessary to distinguish between the “instances” of a clabject and the “offspring” of a clabject. The instances of a clabject exist at the level immediately below that clabject, and can be direct or indirect instances. The offspring of a clabject, on the other hand, are all clabjects in the transitive closure over the “classifies” relationship starting with the subject. In other words, the set of offspring of a clabject includes all the instances of the clabject plus all the instances of those instances, and so on. The set of “direct offspring” is the set of all direct instances, plus all direct instances of those direct instances and so on recursively. In contrast to regular offspring, all indirect instances (at any depth) are excluded.

4.1 Level-Specific Constraints

Level-specific constraints are constraints that restrict the properties of clabjects at one specific level in the deep model relative to the origin clabject. The "scope" of the constraint then just comprises this single level. As explained above, we avoid the term "context" since it may lead to confusion because of the implicit universal quantification semantics of OCL constraints. In general, three cases can be identified – (a) some arbitrary specified level below the starting element, (b) the lowest level containing offspring of the origin and (c) the level containing the starting element itself.

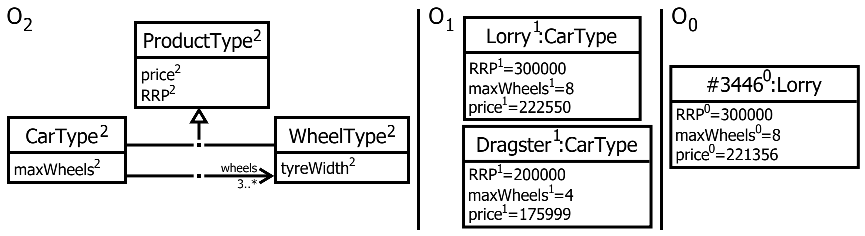


Fig. 6. Level specific constraint example.

When the specified level is defined to be the level immediately below the origin, the scope of the constraint coincides with that implied by an OCL "context". An example using the model elements of Figure 6 is shown in Constraint 6. The constraint ensures that all instances of *CarType* have an RRP between 10k and 400k, but specifically does not target the RRP values of cars. The scope of the level is specified in brackets after the origin keyword. Here, the "1" specifies that the relative distance of the scope to the level on which the constraint has been defined is one. Obviously, this kind of single level scoping also provides the option to evaluate the constraint on any arbitrary level below the origin.

Constraint 6 *The recommended retail price must be between 10k and 400k*

origin*(1) CarType*
self.RRP > 10000 and self.RRP < 400000

The second scoping category defines constraints that apply to the lowest-level containing offspring of the origin clabject without making an explicit reference to the respective instantiation depth. The constraint given in Constraint 7 requires that all instances of *CarType*, at the bottom level have a *price* attribute which is no more than 80% of the RRP. This is specified using the symbol "_" for the scope of the constraint.

Constraint 7 *The price actually paid for a car shall be no more than 80% of the recommended retail price*

***origin**(_) CarType*
*self.price <= self.RRP * 0.8*

The last category represents constraints which cannot be supported in existing constraint languages even though the level they operate on (i.e. the scope) "exists" in traditional modeling approaches. This is the level of the origin element itself. For this reason we refer to this category of constraint as "intra-level" constraints. Such constraints cannot be expressed in traditional environments because in OCL the constrained elements always occupy the instance level while the starting element (referred to as the context in OCL) always occupies the type level.

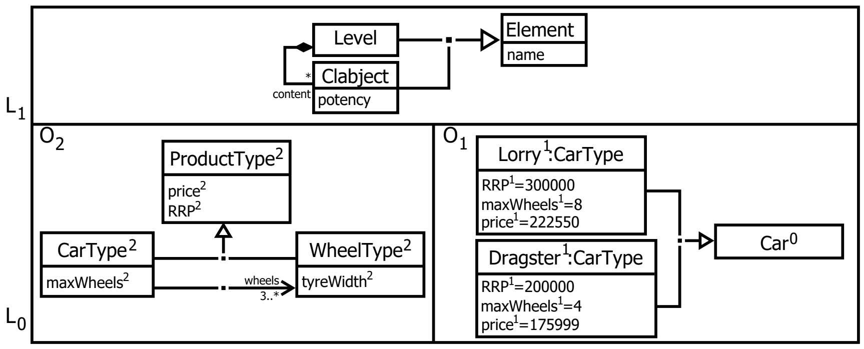


Fig. 7. Intra-level constraint example.

Intra-level Constraints This kind of constraint expresses restrictions on model content that resides on the same level as the origin clabject. An example of such a constraint is Constraint 8 which can be considered to enforce one aspect of the power type pattern [9] on the example displayed in Figure 7.

Constraint 8 *Subtypes of Car with potency higher than 0 must be instances of CarType (cf. PowerType Pattern)*

***origin**(0) Car*
self.getSubclasses() implies forAll(c |
    c._l_ potency > 0 implies c.isTypeOf(CarType))

Here, all subtypes of *Car* are required to also be instances of *CarType*. Such a constraint cannot be reasonably expressed at the level of *CarType* since it would be attached to *CarType* but would have to explicitly restrict its applicability to subtypes of *Car*. If the same or a similar constraint is applied to all subtypes of another superclass, e.g., *CarPrototype* then, in the absence of intra-level constraints, it would have to be attached to *CarType* as well with the respective applicability restriction in place. Such an approach would become unwieldy over time and its complexity is simply an expression of mis-locating the constraint(s).

Constraint 9 is another example of an intra-level constraint. It requires all non-concrete subclasses of a specific class (here *Car* in Figure 7) to have more attributes than *Car*, i.e., to exclude "hollow" subclasses. Again, it may not make sense to exclude hollow subclasses for all instances of *Car's* type and it seems to be unwarranted to force users to define a dedicated "≪non-hollow≫"-stereotype that is then applied only to *CarType*. In other words, we believe there are applications for "one-off" constraints that only apply to a particular instance, without having relevance to other instances of the same type.

Constraint 9 *Subtypes of Car with potency higher than 0 must add attributes*

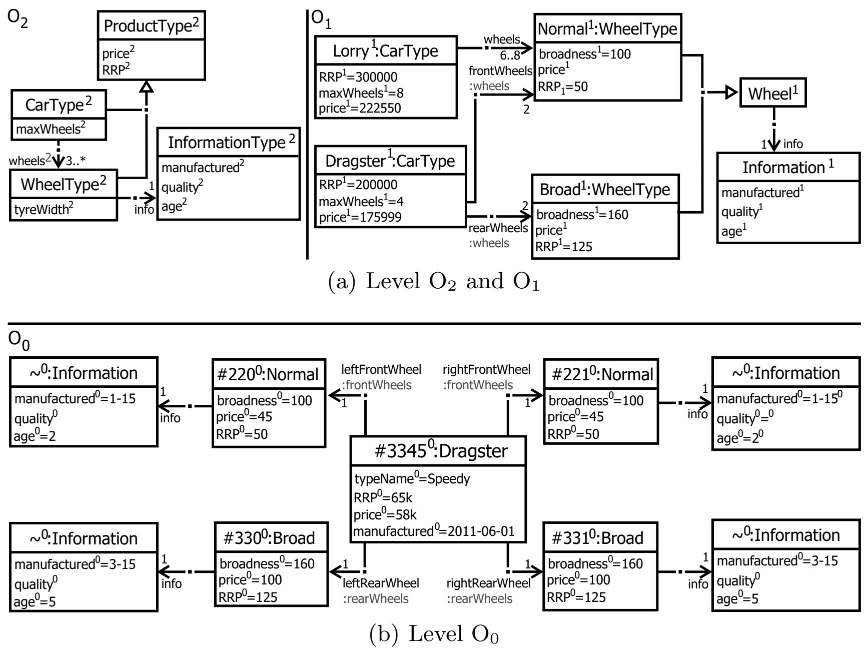
```
origin(0) Car
self.getSubclasses() implies forAll(c |
    c._l_potency > 0 implies
    c.attributes→size() > self.attributes→size())
```

To fully support deep constraints, further capabilities are needed. First, vertical access to offspring and types at any level may be necessary to ensure certain kinds of consistency constraints. The functions isDirectOffspringOf() and isIndirectOffspringOf() could be used like their corresponding OCL statements but based on the notion of offspring rather than instances. Second, it is necessary to support horizontal, intra-level navigation to other elements by using navigation paths that are defined anywhere at levels above. A respective approach has been explored in [4].

4.2 Level-Spanning Constraints

The common property of the previous category of scopes is that the constrained clabjects all occupy one specific ontological level. It is also possible, depending on the flavour of deep modeling in use, to also define constraints that span more than one ontological level. Such constraints would therefore have a scope greater than one in our terminology. This makes sense if the underlying deep modeling language supports uniform (ontological) attributes which, at all levels, possess a name, a type and a value. When an attribute has potency 1 or higher, the value of the attribute is interpreted as a default value for the corresponding attributes of the clabject's instances. In those cases where the constraint for the default values is the same as for the ultimate values, it is beneficial to interpret a constraint as being applicable over more than one level. In this case two situations seem to be useful in practice: (a) a specified arbitrary range of levels or (b) the level of the starting clabject and all its offspring.

In the first kind of level-spanning constraint, the range of levels over which the constraint should apply is explicitly specified relative to the origin clabject.



(a) Level O_2 and O_1

(b) Level O_0

Fig. 8. Level-spanning constraint example. Gray elements are extensions to the deep modeling approach as described in [4].

The constraint shown in Constraint 10 constrains all *WheelType* instances in Figure 8 occupying the following two levels (O_1 and O_0) to be not older than 24 months.

Constraint 10 *All offspring of wheelTypes, over two levels, are not allowed to be older than 24 months*

***origin**(1..2) WheelType*
self.info.age $\leq$ *24*

The second kind of level-spanning constraint is a special case of the first kind. It simply uses the maximum scope possible, i.e., the level of the origin itself up to the lowest level containing offspring. Constraint 11 constrains all offspring of *CarType*, and *CarType* itself, to have a (default-) price of at least 10000.

Constraint 11 *A (default-) price for a car must exceed 10000*

***origin**(0..._) CarType*
self.price $\geq$ *10000*

An example of a level-spanning constraint that includes intra-level navigation is shown in Constraint 12. The constraint specifies that, for security reasons, a *Dragster* must have wheels which are not older than seven months. As can be seen in Figure 8, a *Dragster* is connected to its wheels using two different types of connections. There is one type for the front wheels and one type for the rear wheels, as dragsters require different wheel types depending on their location. For the purposes of Constraint 12, however, it would be desirable to navigate to all wheels in a convenient way, i.e., abstract away from the fact that there are front wheels and rear wheels. Thus, in version (a) using the DeepOCL syntax [10], the statement $CarType$ makes all navigations of its type available, here *wheels*. The MetaDepth version presented in [15] is shown in (b) which uses the "references" linguistic method to get the instances of the *wheel* references and access their value using the *value* method. The version in (c) displays the navigation semantics presented in [4]. The latter relies on the fact that the "wheels" role was introduced with potency two and hence makes the connection available at level O_0, plus the fact that "frontWheels" and "rearWheels" are declared to be instances of "wheels" (cf. Figure 8 (a)).

Constraint 12 *The wheels of Dragsters are not allowed to be older than 7 months*

```
origin(1) Dragster
(a) self.$CarType$.wheels.info implies forAll(age <= 7)
(b) self.references("wheels") implies
        forAll(r | self.value(r).info.age <= 7)
(c) self.wheels.info→forAll(age <= 7)
```

Default Scope for Constraints As constraints in a deep constraint language may have a variety of scopes, the question arises as to which default scope should be assumed in case the modeler does not provide one. Default values in general, can reduce the complexity of a specification and relieve the modeler from explicitly providing a value that they would typically use in most cases. Requirements for a good default choice include

– frequency of occurrence. Making the value that occurs the most implicit, has the largest effect on specification reduction and will also most frequently relieve the modeler from providing it.
– robustness against change. Typical modifications to models should ideally not require the replacement of a default value with a different, specific one.
– generality across different host languages. One and the same deep constraint language may be applicable to a variety of different multi-level host languages. It would be desirable to be able to interpret constraints independently of the host language (e.g., MOF vs UML) and hence the default scope should ideally be always the same.
– validity of existing assumptions. It is desirable to make any extension – such as introducing depth to constraints – conservatively, i.e., not invalidate existing specifications if they do not need more than two levels. Changing the default value from a two-level technology (such as standard OCL) to another value in a multi-level context should only be done with a good justification in order to avoid gratuitously breaking the conservative extension property.

Currently there is no significant body of multi-level constraints from which solid frequency figures could be derived. Anecdotal evidence suggests, however, that most attributes follow a traditional pattern, i.e., are defined at a certain level and receive a value at a lower level. In other words, the default scope should probably not include “0”, i.e., the current level.

The robustness argument suggests that it is probably not advisable to have the default scope include the lowest-level offspring (i.e., “_”). Consider the addition of a new lower level of used book copies to an existing model of (new) books. An existing constraint on prices for new books, should not be automatically re-targeted to used book copy prices, as the latter prices will obey different rules than new book prices.

Not all multi-level languages support the assignment to attributes at all levels at which they (implicitly) occur. This suggests that a default scope should probably not address multiple levels at once.

Finally, current OCL constraints assume an origin of “(1)” implying that OCL experts would have the least effort to adapt to it as a default scope. In combination, all prior observations suggest that the scope “(1)” has the highest appeal. However, further research is necessary to make a more informed choice. For example, there is a need to ascertain actual frequency of use figures, a need to observe and record typical model changes, and track the development of future multi-level languages to re-assess commonalities and differences.

5 Conclusion

In this paper we have identified the additional opportunities and challenges that arise when expressing constraints in the context of deep modeling. It is possible to write types of constraints on deep models that cannot be specified in traditional, two-level modeling environments (e.g. UML/OCL), such as hybrid constraints, deep constraints (targeting remote levels and/or spanning two or more levels) and intra-level constraints (supporting “one-off” constraints). These constraints do not increase the expressiveness compared to a regular two-level constraint language, as we do not introduce constraints over constraints and our proposed mechanisms could be translated into a two-level scheme. However, we believe that our proposed mechanisms are important for allowing modelers to adequately express constraints in a multi-level context. Expressing such constraints in a concise yet unambiguous way requires new concrete syntax and default conventions that do not yet exist. Some initial ideas for these have been presented in this paper, but it was not the goal to define or propose a definitive DCL. Nor was it the goal of this paper to describe precisely what kinds of requirements a DCL should aim to support, since the optimal language from a pragmatic perspective may not need to support every conceivable kind of constraint that can be imagined if no practical use case exists for them. The main goal of the paper was to characterize the kind of constraints that may make sense in the context of deep modeling and to provide a conceptual framework / terminology for discussing them.

References

1. Aspectocl: Extending ocl for crosscutting constraints. In: Taentzer, G., Bordeleau, F. (eds.) Modelling Foundations and Applications, Lecture Notes in Computer Science, vol. 9153, pp. 92–107. Springer International Publishing (2015), http://dx.doi.org/10.1007/978-3-319-21151-0_7
2. Ahmed, A., Vallejo, P., Kerboeuf, M., Babau, J.P.: Cdmcl, a specific textual constraint language for common data model. In: OCL 2014 – OCL and Textual Modeling: Applications and Case Studies (2014)
3. Atkinson, C., Gerbig, R.: Melanie: Multi-level modeling and ontology engineering environment. In: Proceedings of the 2nd International Master Class on Model-Driven Engineering: Modeling Wizards. pp. 7:1–7:2. MW '12, ACM, New York, NY, USA (2012)
4. Atkinson, C., Gerbig, R., Kühne, T.: A unifying approach to connections for multi-level modeling. Models'15, Ottawa, Canada (2015)
5. Atkinson, C., Gerbig, R., Tunjic, C.: Enhancing classic transformation languages to support multi-level modeling. Software & Systems Modeling 14(2), 645–666 (2015)
6. Atkinson, C., Kühne, T.: Concepts for comparing modeling tool architectures. In: Briand, L., Williams, C. (eds.) Model Driven Engineering Languages and Systems, Lecture Notes in Computer Science, vol. 3713, pp. 398–413. Springer Berlin Heidelberg (2005), http://dx.doi.org/10.1007/11557432_30
7. Fowler, M.: Domain-specific languages. Pearson Education (2010)
8. Gogolla, M., Sedlmeier, M., Hamann, L., Hilken, F.: On metamodel superstructures employing uml generalization features. In: MULTI 2014–Multi-Level Modelling Workshop Proceedings. p. 13 (2014)
9. Gonzalez-Perez, C., Henderson-Sellers, B.: A Powertype-based Metamodelling Framework. Software & Systems Modeling 5(1), 72–90 (2006)
10. Kantner, D.: Specification and Implementation of a Deep OCL Dialect. Master's thesis, University of Mannheim (2014), https://ub-madoc.bib.uni-mannheim.de/37143/
11. Kolovos, D., Paige, R., Polack, F.: The epsilon object language (eol). In: Rensink, A., Warmer, J. (eds.) Model Driven Architecture Foundations and Applications, Lecture Notes in Computer Science, vol. 4066. Springer Berlin Heidelberg (2006)
12. Kolovos, D., Paige, R., Polack, F.: On the evolution of ocl for capturing structural constraints in modelling languages. In: Abrial, J.R., Glsser, U. (eds.) Rigorous Methods for Software Construction and Analysis, Lecture Notes in Computer Science, vol. 5115, pp. 204–218. Springer Berlin Heidelberg (2009), http://dx.doi.org/10.1007/978-3-642-11447-2_13
13. de Lara, J., Guerra, E.: Deep meta-modelling with metadepth. In: Proceedings of the 48th international conference on Objects, models, components, patterns. pp. 1–20. TOOLS'10, Springer-Verlag, Berlin, Heidelberg (2010)
14. de Lara, J., Guerra, E., Cuadrado, J.: Model-driven engineering with domain-specific meta-modelling languages. Software & Systems Modeling 14(1) (2015), http://dx.doi.org/10.1007/s10270-013-0367-z
15. Lara, J.D., Guerra, E., Cuadrado, J.S.: When and how to use multilevel modelling. ACM Trans. Softw. Eng. Methodol. 24(2), 12:1–12:46 (Dec 2014), http://doi.acm.org/10.1145/2685615
16. Störrle, H.: Mocql: A declarative language for ad-hoc model querying. In: Van Gorp, P., Ritter, T., Rose, L. (eds.) Modelling Foundations and Applications, Lecture Notes in Computer Science, vol. 7949. Springer Berlin Heidelberg (2013), http://dx.doi.org/10.1007/978-3-642-39013-5_2

An OCL-based Bridge from Concrete to Abstract Syntax

Adolfo Sánchez-Barbudo Herrera[1], Edward Willink[2], Richard F. Paige[1]

[1] Department of Computer Science, University of York, UK.
{asbh500, richard.paige}_at_york.ac.uk
[2] Willink Transformations Ltd. ed_at_willink.me.uk

Abstract. The problem of converting human readable programming languages into executable machine representations is an old one. EBNF and Attribute grammars provide solutions, but unfortunately they have failed to contribute effectively to model-based Object Management Group (OMG) specifications. Consequently the OCL and QVT specifications provide examples of specifications with significant errors and omissions. We describe an OCL-based internal domain specific language (DSL) with which we can re-formulate the problematic parts of the specifications as complete, checkable, re-useable models.

1 Introduction

The Object Management Group (OMG) is a consortium whose members produce open technology standards. Some of these target the Model-Driven Engineering (MDE) community. OMG provides the specifications for languages such as UML [1], MOF [2], OCL [3] and QVT [4].

The specifications for textual languages such as OCL and QVT define a textual language and an information model using:

- an EBNF grammar to define the textual language
- a UML metamodel to define the abstract syntax (AS) of the language

The textual language is suitable for users and for source interchange between compliant tools. The information model facilitates model interchange between producing tools such as editors or compilers and consuming tools such as program checkers or evaluators.

On one hand, textual language designers intend to create compact grammars, without limiting the textual language capabilities and conciseness for end users. On the other hand, model information designers intend to create well designed abstract syntaxes to facilitate the model information adoption by producing and consuming tools. These intentions are not normally aligned: unless we sacrifice the interests of any of the mentioned stakeholders, we get the situation in which we have a big gap between the textual language grammar and the model information, and additional conversions between the different involved data structures are required.

Therefore, the conversion between these two representations must also be specified and may make use of an additional intermediate concrete syntax (CS) metamodel whose elements correspond to the productions and terminals of the textual language grammar[3]. OMG specifications tend to provide concise textual languages grammars, and well designed AS metamodels, without compromising one in favour of the other. In consequence, CS to AS conversions are defined in some OMG specifications, however, as we will see along this paper, there is room for improvement.

1.1 The OMG specification problem

The OCL [3] and QVT [4] specifications define four languages, OCL, QVTc (Core), QVTo (Operational Mappings), and QVTr (Relations). The specifications all provide fairly detailed grammars and metamodels of their respective abstract syntaxes.

Unfortunately the grammar to AS conversion is poorly specified.

In OCL, a CS is provided and the grammar is partitioned into ambiguous productions for each CS element. Semi-formal rules define the grammar to CS correspondence, the CS to AS correspondence, name resolution and disambiguation.

QVTr has a single coherent grammar to accompany its CS and similar semi-formal rules.

QVTc has a single grammar but no CS and no semi-formal rules.

QVTo similarly has a single grammar, but no CS and no semi-formal rules. Instead, notation sections suggest a correspondence between source text and AS elements by way of examples.

Since none of the conversions are modeled, tools cannot be used to check the many details in the specifications. As a result, the major omissions identified above are augmented by more subtle oversights and inaccuracies. The specifications fail to provide the complete, consistent and accurate details to help tool vendors to provide compliant implementations of the text to AS conversions.

1.2 Our solution

The intermediate CS metamodel is close to the grammar, and it can be automatically generated by modern Annotated EBNF tooling such as Xtext. It is in the CS to AS conversion that greater challenges arise.

In this paper, we take inspiration from the substantial semi-formal exposition of the OCL conversions (Clause 9.3 of [3]) and introduce a fully modeled CS2AS bridge. The models can be used to variously check and even auto-generate a consistent specification and also to auto-generate compliant tooling. In addition to conventional CS and AS metamodels, we introduce new CS2AS mapping models, name resolution models and CS disambiguation models. We demonstrate

[3] Modern language workbenches, such as Xtext, can automatically generate the CS metamodel from their input grammars

how OCL itself can be used to provide a suitable internal DSL for these new models.

The paper is structured as follows. Section 2 presents an example to introduce the grammar and metamodels. Section 3 demonstrates the semi-formal solution adopted by the OCL specification. Section 4 explains the proposed solution, i.e. an OCL-based internal DSL. Section 5 describes related work and Section 6 talks about the current shortcomings of the approach. Section 7 outlines some future work, including how tool implementors can benefit from the internal DSL. Finally, Section 8 concludes.

2 Example

Our first example is a collection literal expression. This provides a simple example of the grammars and models in use. In Section 3 we show the semi-formal usage of these concepts by the OCL specification. In Section 4 we provide a contrast with our fully-modeled internal DSL solution. This example is too simple to demonstrate more than the CS2AS characteristics of our solution. We therefore introduce a further more relevant example later.

The listing in Figure 1 is an example of a *collection literal expression* comprising three comma-separated collection literal parts. The adjacent diagram shows the corresponding AS metamodel elements. *CollectionLiteralExp* contains many abstract *CollectionLiteralParts*. *CollectionItem* and *CollectionRange* are derived to support the two cases of a single value or a two-ended integer range. The example text must be converted to instances of the AS metamodel elements.

```
Sequence{1, 1+1, 3..9+1}

-- equivalent to:

-- Sequence{1,2,3,4,5,
--      6,7,8,9,10}
```

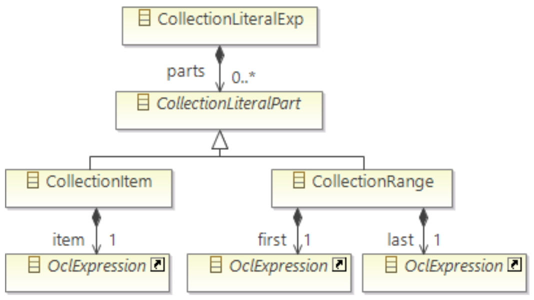


Fig. 1: CollectionLiteralPart Example and partial AS Metamodel

The listing in Figure 2 shows the EBNF grammar that parses a *collection literal part* as a *CollectionLiteralPartCS* comprising one direct *OclExpressionCS* or a *CollectionRangeCS* comprising two *OclExpressionCSs*. The adjacent diagram shows the intermediate CS model, which is similar to the AS but which omits a 'redundant' *CollectionItemCS* preferring to share a single/first expression from the non-abstract *CollectionLiteralPartCS*.

```
CollectionLiteralPartCS:
  OclExpressionCS | CollectionRangeCS

CollectionRangeCS:
  OclExpressionCS '..' OclExpressionCS
```

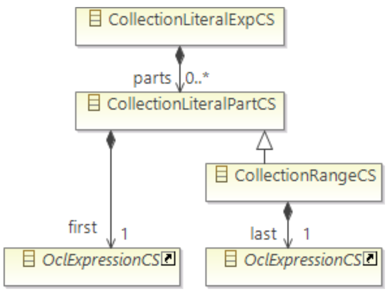
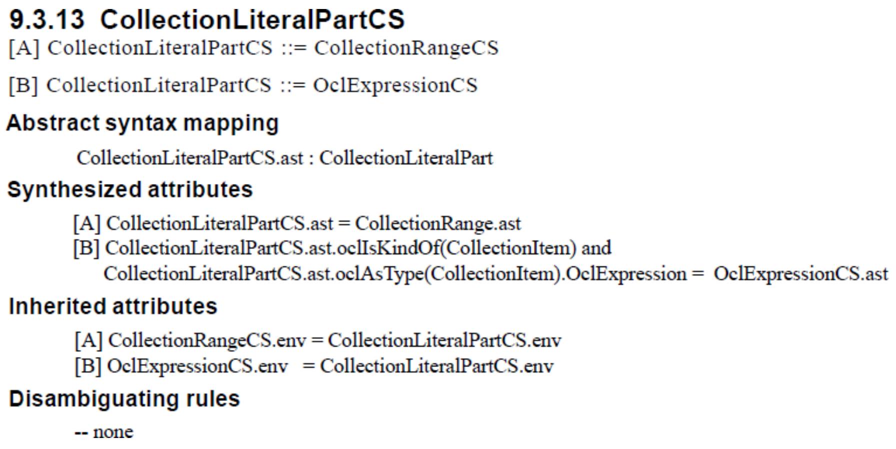


Fig. 2: CollectionLiteralPartCS Grammar and partial CS Metamodel

3 Semi-formal solution: OCL Clause 9.3

The OCL specification provides a full attribute grammar in which inherited and synthesized attributes are used to describe how the AS is computed from the CS. Figures 3 and 4 shows our first example. The specification uses OCL expressions to express how the different attributes are computed.



9.3.13 CollectionLiteralPartCS

[A] CollectionLiteralPartCS ::= CollectionRangeCS

[B] CollectionLiteralPartCS ::= OclExpressionCS

Abstract syntax mapping

CollectionLiteralPartCS.ast : CollectionLiteralPart

Synthesized attributes

[A] CollectionLiteralPartCS.ast = CollectionRange.ast
[B] CollectionLiteralPartCS.ast.oclIsKindOf(CollectionItem) and
CollectionLiteralPartCS.ast.oclAsType(CollectionItem).OclExpression = OclExpressionCS.ast

Inherited attributes

[A] CollectionRangeCS.env = CollectionLiteralPartCS.env
[B] OclExpressionCS.env = CollectionLiteralPartCS.env

Disambiguating rules

-- none

Fig. 3: OCL specification for CollectionLiteralPartCS to CollectionLiteralPart

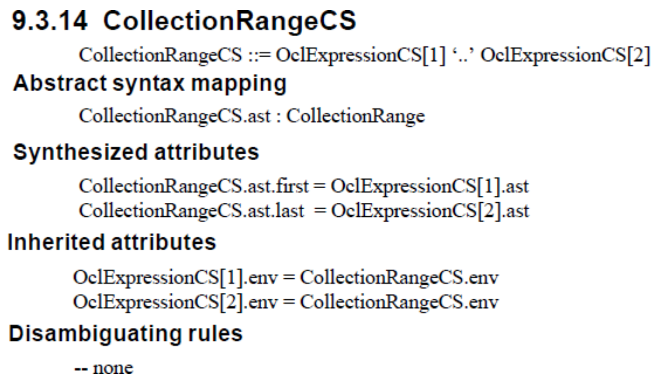
The first section defines the EBNF production(s). The example merges two alternate productions and so many of the rules have an `[A]` or `[B]` prefix to accommodate the alternative rules.

The AS mapping declares the type of the resulting AS element as the type of a special property of the CS element: *ast*.

The synthesized attributes populate the AS element using an assignment for `[A]`. The more complex `[B]` worksaround OCL 2.4's inability to construct a *CollectionItem* by imposing constraints on a hypothetical *CollectionItem*.

The inherited attributes contribute to the name resolution by flowing down an *Environment* hierachy of all available name-element pairs from parent to child nodes using another special CS property: *env*. In this case all names visible in the parent are passed without modification to the children.

The disambiguating rules provide guidance on the resolution of ambiguities. In this simple example, there is no ambiguity.


9.3.14 CollectionRangeCS

CollectionRangeCS ::= OclExpressionCS[1] '..' OclExpressionCS[2]

Abstract syntax mapping

CollectionRangeCS.ast : CollectionRange

Synthesized attributes

CollectionRangeCS.ast.first = OclExpressionCS[1].ast
CollectionRangeCS.ast.last = OclExpressionCS[2].ast

Inherited attributes

OclExpressionCS[1].env = CollectionRangeCS.env
OclExpressionCS[2].env = CollectionRangeCS.env

Disambiguating rules

-- none

Fig. 4: OCL specification for CollectionRangeCS to CollectionRange

The rules for collection range follow a similar pattern. There is now just one grammar production whose two *OclExpressions* are distinguished by `[1]` and `[2]` suffixes. The synthesized attributes have two properties to populate.

3.1 Critique

The presentation comes quite close to specifying what is needed, but uses an intuitive mix of five sub-languages without any tool assistance. In Figure 3, the typo whereby *CollectionItem::OclExpression* rather than *CollectionItem::item* is used in the final line of the synthesized attributes has gone unreported for over 10 years.

The lack of tooling also obscures the modeling challenge for the inheritances between *CollectionLiteralPartCS*, *CollectionRangeCS* and *OclExpressionCS*. The `[B]` grammar production in Figure 3 requires *OclExpressionCS* to inherit from *CollectionLiteralPartCS*, if *CollectionLiteralPartCS* is to be the polymorphic type of any collection literal part in the CS.

The lack of any underlying models makes it impossible for tool vendors to re-use the rules. Tool vendors must transcribe and risk introducing further errors.

4 Modeled Solution: CS2AS internal DSL

The critique of the semi-formal exposition highlights the lack of checkable or re-useable models. In this section we formalize the semi-formal approach using a DSL to declare the bridge between the CS and the AS of a language. The DSL is internal [5] and uses only facilities proposed for OCL 2.5. The DSL constrains the use of the general purpose OCL language to define a set of idioms that express CS2AS bridges.

Our rationale for choosing OCL as the host language is as follows:

- OMG specifications have a problem with bridging the CS to AS gap, so we would like an OMG-based solution.
- OCL contains a rich expression language which can provide enough flexibility to express non trivial CS2AS bridges in a completely declarative way.
- Other OMG related languages could be considered (such as one of the QVT languages), however OCL is a well known OMG language and is the basis of many others. A QVT practitioner inherently knows OCL but not vice-versa.

Instances of the internal DSL take the form of Complete OCL documents and can be maintained using Complete OCL tools [6]. Multiple documents can be used to partition the specification into modules to separate the distinct mapping, name-resolution, and disambiguation concerns of the CS2AS bridge.

4.1 Shadow Object Construction

The internal DSL uses the proposed[4] side-effect-free solution to the problem of constructing types in OCL. This avoids the need for the hypothetical objects used by the semi-formal approach. The proposed syntax re-uses the existing syntax for constructing a Tuple. The *Tuple* keyword is replaced by the name of the type to be constructed. A `Complex` number with `x` and `y` parts might therefore be constructed as `Complex{x=1.0,y=2.0}`.

4.2 CS2AS mappings

In this subsection we explain the main CS2AS mappings description language. We start by introducing an instance of the language so that the reader can have an indication of the DSL used to describe the bridge. Listing 1.1 corresponds to the CS2AS description of the OCL constructs introduced in Section 2. The listing should be contrasted with the semi-formal equivalent in Figures 3 and 4.

```
context CollectionLiteralPartCS
def : ast() : ocl::CollectionLiteralPart =
  ocl::CollectionItem {
       item = first.ast(),
       type = first.ast().type
      }

context CollectionRangeCS
def : ast() : ocl::CollectionRange =
  ocl::CollectionRange {
       first = first.ast(),
       last = last.ast(),
       type = first.ast().type.commonType(last.ast().type)
      }
```

Listing 1.1: CS2AS bridge for CollectionLiteralPart and CollectionRange

The mapping is described by defining the *ast()* operation on a CS element. The 'abstract syntax mapping' and 'synthesized attributes' of the semi-formal approach are modeled by the shadow construction of the appropriate AS type

[4] Shadow object construction was called type construction in the Aachen report [7]

and initialization of its properties. (The initialization includes the *type* property omitted by the OCL specification.)

Declarativeness: An important characteristic of the DSL is that it comprises declarative OCL constraints. The OCL constraints specify only true correspondences between AS and CS after a valid conversion. In a scenario of executing the proposed CS2AS descriptions, discovery of a suitable order in which to perform CS to AS conversions requires an implementing tool to analyze the OCL constraints and exploit their inter-dependencies. (This was also the unstated policy of the semi-formal approach.) An automated analysis is desirable since they are almost too complicated for an accurate manual formulation as a multi-pass conversion.

Operations: The CS2AS bridge is described using operation definitions. The underlying rationale is that operation definitions on a potentially complex class hierarchy of the CS can be overridden. Due to this overriding mechanism, we provide some flexibility to cope with language extensions such as QVT. The operation name is not relevant, but we propose the name *"ast"* since it is aligned with the name used in the attribute grammar exposed in the OCL specification.

Shadow object construction: Shadow object constructions express how AS elements are constructed and how their properties are initialized.

Operation Calls: To compute properties of any AS element, we need to access the AS elements to determine a CS to AS correspondence. Since *ast()* is a side-effect-free query, we may call *ast()* as many times as necessary to obtain the appropriate AS element. For example, at line 4, in order to initialize the *CollectionItem::item* property, we use the *ast()* to obtain the *OclExpression* corresponding to the *first OclExpressionCS* of the context *CollectionLiteralPartCS*.

Self-contained: With the goal in mind of using the proposed internal DSL to rewrite part of the OMG specifications, the declaration of the CS2AS bridge for a particular CS element is complete and self-contained. The computations for all non-default-valued properties of the corresponding AS element are expressed directly in the shadow type expression since there is no constructor to share inherited computations.

Reusable computations: Having OCL as the host language for our internal DSL, we can factor out and define more complex and reusable expressions in new operation definitions. The operations can be reused, by just introducing operation call expressions, across the different computations of the AS element properties. For example, a t line 13 of Listing 1.1, *commonType* is a reusable operation to compute the common supertype of source and argument types.

4.3 Name resolution description

In this subsection, we explain how name resolution is described when defining CS2AS bridges by the means of our OCL-based internal DSL. In a name resolution activity we can typically find two main roles:

– a producer provides a name-to-element map for all possible elements in its producing scope.

– a consumer looks up a specific element corresponding to a name in its consuming context

Our previous example had no need to resolve names, so we will now introduce a new example with a name producer and a consumer. The listing in Figure 5 is an example of a *let expression* that declares and initializes a variable named *var* for use within the *'in'* of the *let expression*. In this example the *'in'* comprises just a *variable expression* that references *var*. The adjacent diagram shows the corresponding AS metamodel elements. A *LetExp* contains the produced *Variable* and an arbitrary *OclExpression 'in'*. For our simple example the *'in'* is just a *VariableExp*. The complexity of the example lies in the initialization of the consuming *VariableExp.referred Variable* to reference the producing *LetExp.variable*.

```
let var : String = 'something'
in var
```

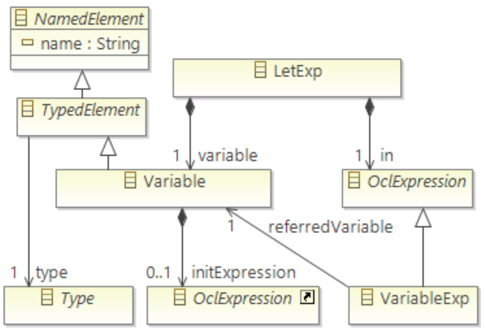


Fig. 5: LetExp/VariableExp Example and partial AS Metamodel

Figure 6 shows the corresponding grammar and CS definitions[5]. A *LetExpCS* contains a *VariableDeclarationCS* and *OclExpressionCS* which for our example is just a *VariableExpCS*.

```
LetExpCS:
  'let' VariableDeclarationCS
  'in' OclExpressionCS

VariableDeclarationCS:
  simpleName (':' TypeCS)?
  ('=' OclExpressionCS)?

VariableExpCS:
  simpleName | 'self'
```

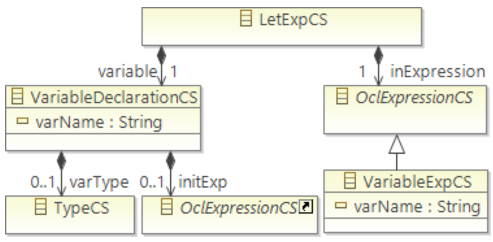


Fig. 6: LetExpCS/VariableExpCS Grammar and partial CS Metamodel

In typical programming languages every use of a variable has a corresponding declaration. The variable declaration is the producer of a name-to-variable mapping. The variable usage consumes the variable by referencing its name.

[5] The complexity of multi comma-separated variables has been removed, because it is not needed to explain how name resolution is described in our interal DSL

Name resolution searches the hierarchy of producing contexts that surround the consuming context to locate a name-element mapping for the required name.

In our example, the required cross-reference in the AS is represented in the CS by the distinct *VariableDeclarationCS.varName* and *VariableExpCS.varName* properties. These are both parsed with the value *var* and so, when consumption of the *VariableExpCS.varName* is analyzed, the analysis must discover the corresponding *VariableDeclarationCS.varName* production.

The semi-formal approach adopted by the OCL specification re-uses the containment hierarchy of the CS as the scope hierarchy for its 'inherited attributes'. The name-to-element mappings are maintained in an *Environment* hierarchy. The mappings flow down from the root CS element to all the leaf elements which accumulate additional name-to-element mappings and/or nested environments at each intermediate CS element in the CS tree.

In Section 3 we saw the very simple unmodified flow-down for a *Collection-LiteralPart*. The equivalent exposition for a *LetExp* in the OCL specification is complicated by performing the CS2AS mapping of multiple comma-separated let-variables with respect to the CS rather than the AS. We therefore present its logical equivalent in Listing 1.2.

```
LetExpCS ::= let VariableDeclarationCS in OclExpressionCS

VariableDeclarationCS.env = LetExpCS.env
OclExpressionCS.env = LetExpCS.env.nestedEnvironment().addElement(
     VariableDeclarationCS.ast)
```

Listing 1.2: Semi-formal LetExpCS equivalent

The environment of the LetExpCS is passed unchanged to the VariableDeclarationCS so that name resolution within the VariableDeclarationCS initializer sees the same names as the LetExpCS.

The environment for the OclExpressionCS is more interesting. A nested Environment is created containing the name-to-variable mapping for the let-variable. The use of a nested environment ensures that the let-variable name occludes any same-named mapping in the surrounding environment.

Our modeled approach is very similar but re-uses the AS tree rather than the CS tree as the scope hierarchy. The rationale is that we are interested in looking up AS elements for which we might not have the corresponding CS (e.g OCL standard library or user model elements – classes, properties, operations, etc. –).

```
context OclAny
def : env : env::Environment =
  if oclContainer() <> null
  then oclContainer().childEnv(self)
  else env::Environment{}
  endif

def : childEnv(child : OclAny) : env::Environment =
  env

context LetExp
def : childEnv(child : OclAny) : env::Environment =
  if child = variable
  then env
```

```
15    else env.nestedEnv().addElement(variable)
16    endif
```

Listing 1.3: Name resolution producers

Listing 1.3 presents the name resolution description written in our OCL-based internal DSL. Line 2 declares an *env* property to hold the immutable *Environment* of the AS element. *env* is initialized by a containment tree descent that uses *oclContainer()*[6]. Line 5 provides an empty environment at the root, otherwise Line 4 uses *childEnv(child)* to request the parent to compute the child-specific environment.

The default definition of *childEnv(child)* on lines 8-9 flows down the prevailing environment to all its children. This can be inherited by the many AS elements that do not enhance the environment.

The non-default override of *childEnv(child)* for LetExp on lines 12-16 uses the *child* argument to compute different environments for the *Variable* and *OclExpression* children. As we saw for the semi-formal approach, the environment for the *Variable* is unmodified. The environment for the *OclExpression* is extended by the addition of the variable in a nested environment.

The environment is exploited by consumers to satisfy their requirement to convert a textual name into the corresponding model element. The conversion comprises three steps

- locate all candidate elements
- apply a filtering predicate to select only the candidates of interest
- return the selected candidate or candidates

The first stage is performed by the environment propagation described above.

The filtering predicate invariably selects just those elements whose name matches a required name. It may often provide further discrimination such as only considering Variables, Properties or Namespaces. For operations, the predicate may also match argument and parameter lists.

The final return stage returns the one successfully selected candidate which is the only possibility for a well-formed conversion. For practical tools a lookup may fail to find a candidate or may find ambiguous candidates and provide helpful diagnostics to the user.

The specification is made more readable if the three stages are wrapped up in helper functions such as *lookupVariable* or *lookupProperty*[7].

List 1.4 shows the polymorphic *ast()* operation to map *VariableExpCS* to *VariableExp*. The *lookupVariable* helper function is used to discover the appropriate variable to be referenced by *referredVariable*.

[6] oclContainer() returns the containing element which is null at the root.

[7] A practical implementation may provide alternative helper implementations that exploit the symmetry of the declarative exposition to search up through the containment hierarchy examining only candidates that satisfy the filtering predicate. This avoids the costs of flowing complete environments down to every AS leaf element where at most one element of the environment is of interest.

```
context VariableExpCS
def : ast() : ocl::VariableExp =
  let variable = ast().lookupVariable(varName)
  in ocl::VariableExp {
      name = varName,
      referredVariable = variable,
      type = if variable = null
           then null
           else variable.type
           endif
   }
```

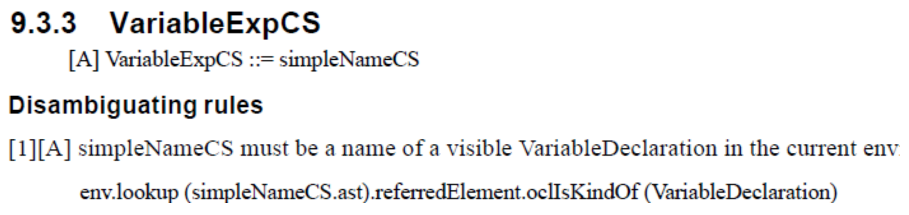
Listing 1.4: CS2AS bridge for VariableExpCS to VariableExp

4.4 Disambiguation

As we commented in the introduction, CS disambiguation is another important concern which needs to be addressed during the CS2AS bridge. To explain the need of disambiguation rules, we consider the simple OCL expression `x.y`.

At first glance, the *'y'* property of the *'x'* variable is accessed using a *property call expression* and a *variable expression*. However *'x'* is not necessarily a variable name. It could be that there is no *'x'* variable. Rather *'x'* may be a property of the implicit source variable, *self*, since the original expression could be a short form for *self.x.y*. Semantic resolution is required to disambiguate the alternatives and arbitrate any conflict.

The OCL specification provides disambiguation rules to 'resolve' grammar ambiguities. Clause 9.1 states : "Some of the production rules are syntactically ambiguous. For such productions disambiguating rules have been defined. Using these rules, each production and thus the complete grammar becomes nonambiguous.". Figure 7 and Figure 8 are extracted from the OCL specification. It can be seen that a *simpleNameCS* with no following `@pre` matches the `[A]` production of a *VariableExpCS* and the `[B]` production of a *PropertyCallExpCS*.



9.3.3 VariableExpCS

[A] VariableExpCS ::= simpleNameCS

Disambiguating rules

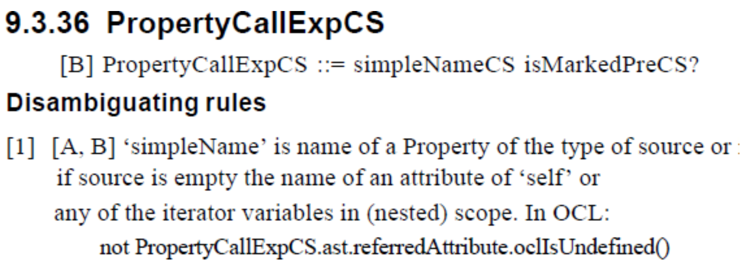
[1][A] simpleNameCS must be a name of a visible VariableDeclaration in the current env

env.lookup (simpleNameCS.ast).referredElement.oclIsKindOf (VariableDeclaration)

Fig. 7: Partial OCL Specification for VariableExpCS to VariableExp

The disambiguation rule for *VariableExpCS* is relatively simple delegating to the *lookup* helper and imposing a constraint that the result must be a *VariableDeclaration*. This is potentially correct, although unfortunately the specification that *VariableDeclaration* is the supertype of *Variable* and *Parameter* is missing.

The disambiguation rule for *PropertyCallExpCS* has some ambiguous wording and many details that do not correspond to the "In OCL". This requires



9.3.36 PropertyCallExpCS

[B] PropertyCallExpCS ::= simpleNameCS isMarkedPreCS?

Disambiguating rules

[1] [A, B] ‘simpleName’ is name of a Property of the type of source or
if source is empty the name of an attribute of ‘self’ or
any of the iterator variables in (nested) scope. In OCL:
not PropertyCallExpCS.ast.referredAttribute.oclIsUndefined()

Fig. 8: Partial OCL Specification for PropertyCallExpCS to PropertyCallExp

intuition by the implementor who may also wish to consider how the rules apply to implicit opposite properties in EMOF.

Both of these disambiguation rules require semantic information which is not available when the syntactic parser requires it. The problem can be avoided by unifying the ambiguous alternatives as unambiguous productions that can be parsed to create a unified CS tree. Once parsing has completed, semantic analysis of the unified CS can resolve the unified elements into their disambiguated forms.

We therefore introduce additional unifying CS elements that can be resolved without semantic information. A unifying *NameExpCS* element replaces *PropertyCallExpCS* and *VariableExpCS*. Figure 9 shows the new unifying CS element.

```
NameExpCS:
 simpleName isMarkedPreCS?
```

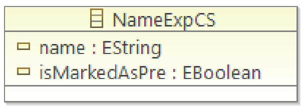


Fig. 9: NameExpCS Grammar and partial CS Metamodel

Listing 1.5 shows the definition for the CS2AS mapping of a *NameExpCS*, in which the *isAVariableExp()* at line 3 is a call of the operation providing the disambiguation rule. The return selects whether a *NameExpCS* is mapped to a *VariableExp* (lines 5-13), otherwise a *PropertyCallExp* (lines 15-23).

```
1  context NameExpCS
2  def : ast() : ocl::OclExpression =
3   if isAVariableExp()
4   then
5     let variable = ast().lookupVariable(name)
6     in ocl::VariableExp {
7        name = name,
8        referredVariable = variable,
9        type = if variable = null
10             then null
11             else variable.type
12             endif
13    }
14  else
15    let property = ast().lookupProperty(name)
16    in ocl::PropertyCallExp {
17       name = name,
18       referredProperty = property,
19       type = if property = null
20             then null
21             else property.type
```

```
            endif
    }
  endif
```

Listing 1.5: CS2AS description for an ambiguous name expression

This approach has the benefit of localizing the disambiguation in the *isAVariableExp()* operation, and so making *VariableExpCS* and *PropertyCallExpCS* redundant. The simple two-way disambiguation decision is shown in Listing 1.6

```
context NameExpCS
def : isAVariableExp() : Boolean =
  let variable = ast().lookupVariable(name)
  in variable <> null
```

Listing 1.6: NameExpCS disambigutation rule

Simple choices such as the various forms of *CollectionLiteralPartCS* can be resolved syntactically. Semantic decisions are required for the unified name example above. The conflicts between the use of parentheses for template arguments, operation calls and iteration calls can be resolved in the same way but with a more complex semantic decision tree.

5 Related work

In this section we briefly discuss how the proposed OCL-based CS2AS bridge relates to previous work. To the best of our knowledge there does not exist a DSL approach based on OMG specifications to describe bridges between CS and AS. The Complete OCL document based approach was introduced in [8] and this paper aims to explain the whole approach (i.e. the internal DSL). Recently, OCLT [9] has been proposed as a functional transformation language to tackle model transformations. Apart from being too novel to be considered in this work, OCLT is not domain specific and it needs additional constructs (e.g. pattern matching) in order to cover more complex transformation scenarios.

We can find languages conceived to sort out the CS2AS bridges in other contexts, i.e in the context of some specific tools. We highlight two of them:

NaBL [10] & Stratego [11]: These are two separate languages for different purposes used by the Spoofax language workbench [12]. The former is used to declare name resolution and the latter to declare syntax rewrites (tree based structure transformations). As a main difference with respect to our approach, these languages are completely unrelated: whereas the former is integrated during the parsing activities in order to resolve cross-references when producing the CS tree, the latter is a general purpose program transformation language further used to obtain the potentially different AS tree. In our approach, we integrate the name resolution language into a further CS2AS activity, provided that the parsing activity first produces a CS tree. As it was commented in Section 4.3, the name lookups are performed on AS elements rather than on CS ones.

Gra2Mol [13]: Gra2Mol is an approach that is closer in objective to the approach presented in this paper. It is a domain specific transformation language conceived to define those bridges, and as our approach does, the name resolution activity is also declared as part of the transformation language. However, whilst their name resolution relies on explicitly specifying a direct search (thus, the name consumer needs to know where the name producer is located in the syntax tree), our approach for specifying name resolution is more declarative based on an independent declaration of name producers and consumer (thus, the name consumer doesn't need to know where the producer is located in the syntax tree). Another difference is that whilst we use OCL as the expression language to express the bridges, they define a structure-shy[8] query language instead. They claim that the usage of their query language is more compact and less verbose when compared to using OCL expressions. However such languages are not suitable from the point of view of OMG specifications. Besides, we can add that structure-shy languages are more error prone or sensitive to changes in the involved metamodels (metamodel evolution): when having a static typed language such OCL, supporting tools can better assist with metamodel evolution.

6 Limitations and shortcomings

From the point of view of the OMG specification, we do not see any limitations of the proposed internal DSL. Having OCL as the host language is a good solution for OMG specifications, because the instances of the DSL can be directly ported to those specifications in order to precisely define the corresponding CS2AS bridges. Likewise, the flexibility and modularity that Complete OCL documents provide has promise in addressing very large CS2AS gaps.

On the other hand, from the final user point of view, i.e the user of the DSL, and specially when comparing with related work, we perceive that having an external DSL fully designed to deal with concepts related to name resolution (e.g. NaBL) or disambiguation may be more convenient. We discuss this further in the next section when talking about future work.

Another shortcoming to mention is that the DSL is based on the concept of shadow type expression, which is not yet part of the OCL specification, although it is planned to be included in the next OCL version (2.5) [7][9]. The number of OCL tools which can currently be used to validate the CS2AS bridges is therefore limited (we are using Eclipse OCL[6] which prototypes some proposed OCL 2.5 features).

7 Ongoing and future work

Apart from using this OCL-based internal DSL to define CS2AS bridges, we are also producing the Java based source code responsible for obtaining AS models from CS ones. This ongoing work follows the line drawn in the introduction which highlights that the CS2AS internal DSL can be exploited by tool implementers. Although in this paper we are unable to go into further detail, we can point

[8] Xpath is an example of this kind of language

[9] It is cited in the report as type construction expression, Section 3.1

the reader out to some JUnit test cases[10] working on small examples, which demonstrate that the instances of the CS2AS internal DSL can be transformed to executable code and perform the CS2AS gap resolution of a language.

In terms of future work, we highlight the following.

– **Definition of CS2AS bridges for OCL and QVT.** We will apply the proposed OCL-based internal DSL to provide complete CS2AS bridge descriptions for the whole OCL and the three QVT languages. We expect these CS2AS bridge specifications to be included as part of the future OCL and QVT specifications. Likewise, we expect auto-generated code from from these bridge specifications to be used in future releases of the Eclipse OCL and QVTd projects. This should eliminate errors attributable to hand-written conversion source code.
– **Incremental CS2AS bridges.** Since generation of code from the declarative CS2AS bridges requires a detailed dependency analysis to identify a valid conversion schedule, we plan to exploit this analysis to synthesize incremental code for use in interactive contexts such as OCL editors. This should improve accuracy and performance dramatically since accurate efficient incremental code is particularly hard to write manually and pessimistic simplifications to improve accuracy are not always sound.
– **Creation of an external DSL.** By bringing together the good aspects of other related languages such as NaBL or Gra2Mol, we plan to create an external DSL and with a higher level of abstraction and more concise than the one presented here, to ease even more the creation of those bridges. This external DSL can embed the OCL expressions language, and the supporting tooling can include a code generator to modularly produce the instances of the internal DSL presented in this paper.
– **Integration with existing language workbenches.** As added value of the DSL and to provide more proofs about how tool vendors may benefit from it (not covered in this paper), we want to exploit the proposed DSL in the context of a modern language workbench called Xtext.

8 Conclusions

We have introduced a Concrete Syntax to Abstract Syntax bridge that is:

– **Sound**. We have shown how intuitive aspects of the current OCL specification are formalized by OCL definitions and faults corrected.
– **Executable**. We can use the dependencies behind the OCL definitions to establish an execution schedule.
– **Extensible**. We can reuse the formalization of the OCL bridge in a QVT bridge.

Our bridge modularizes and separates the specification concerns:

[10] http://git.eclipse.org/c/mmt/org.eclipse.qvtd.git/tree/tests/org.eclipse.qvtd.cs2as.compiler.tests/src/org/eclipse/qvtd/cs2as/compiler/tests/OCL2QVTiTestCases.java

– **Mapping**. An OCL operation hierarchy maps CS artifacts to the AS.
– **Name Resolution**. An OCL operation hierarchy flows the visible names down to the point of access.
– **Disambiguation**. Unified CS artifacts, plus CS disambiguation rules, avoid the need for semantic resolution within a syntactic parser.

Our bridge is currently ready-to-go; it works on test examples. It will now be applied to replace manual tooling in Eclipse OCL and QVT by tooling generated direct from the potential OCL 2.5 specification models.

Acknowledgement. We gratefully acknowledge the support of the UK Engineering and Physical Sciences Research Council, via the LSCITS initiative,

References

1. Object Management Group. Unified Modeling Language (UML), V2.5. OMG Document: ptc/2012-10-24 (`http://www.omg.org/spec/UML/2.5`), 2012.
2. Object Management Group. Meta Object Facility (MOF) Core Specification, V2.4.1. OMG Document: formal/2013-06-01 (`http://www.omg.org/spec/MOF/2.4.1`), 2013.
3. Object Management Group. Object Constraint Language (OCL), V2.4. OMG Document: ptc/2013-08-13 (`http://www.omg.org/spec/OCL/2.4`), January 2013.
4. Object Management Group. Meta Object Facility (MOF) 2.0 Query/View/-Transformation V1.2. OMG Document: ptc/2014-03-38 (`http://www.omg.org/spec/QVT/1.2`), May 2014.
5. Martin Fowler. *Domain-Specific Languages*. Pearson Education, 2010.
6. The Eclipse Foundation. Eclipse OCL. On-Line: `http://projects.eclipse.org/projects/modeling.mdt.ocl`, 2005.
7. Achim D Brucker, Dan Chiorean, Tony Clark, Birgit Demuth, Martin Gogolla, Dimitri Plotnikov, Bernhard Rumpe, Edward D Willink, and Burkhart Wolff. Report on the aachen ocl meeting. In *Proceedings of the MODELS 2013 OCL Workshop*, volume 1092. CEUR Workshop Proceedings, 2014.
8. Adolfo Sánchez-Barbudo Herrera. Enhancing Xtext for General Purpose Languages. In Benoit Baudry, editor, *Proceedings of the Doctoral Symposium at MODELS'14*, volume 1321. CEUR Workshop Proceedings, 2014.
9. Frédéric Jouault, Olivier Beaudoux, Matthias Brun, Mickael Clavreul, and Guillaume Savaton. Towards functional model transformations with ocl. In *Theory and Practice of Model Transformations*, pages 111–120. Springer, 2015.
10. Gabriël Konat, Lennart Kats, Guido Wachsmuth, and Eelco Visser. Declarative name binding and scope rules. In *Software Language Engineering*, volume 7745, pages 311–331. Springer Berlin Heidelberg, 2013.
11. Eelco Visser. Program transformation with stratego/xt. In Christian Lengauer, Don Batory, Charles Consel, and Martin Odersky, editors, *Domain-Specific Program Generation*, volume 3016 of *Lecture Notes in Computer Science*, pages 216–238. Springer Berlin Heidelberg, 2004.
12. Delft University of Technology. Spoofax. On-Line: `http://strategoxt.org/Spoofax`.
13. Javier Luis Cánovas Izquierdo and Jesús García Molina. Extracting models from source code in software modernization. *Software & Systems Modeling*, 13:1–22, 2012.

On the Use of Active Operations for Incremental Bidirectional Evaluation of OCL

Frédéric Jouault and Olivier Beaudoux

Groupe ESEO, Angers, FRANCE,
{*firstname.lastname*}@eseo.fr

Abstract. Many relations between model elements are expressed in OCL. However, tool support to enable synchronization of elements based on OCL-expressed relations is lacking. In this paper, we propose to use active operations in order to achieve incremental execution of some OCL expressions. Moreover, bidirectionality can also be achieved in non-trivial cases.

1 Introduction

Relations between model elements are often expressed as OCL [9] expressions. These may be intra-model relations, which may for instance specify the values of derived features with respect to the values of other features. They may also be inter-model relations, which may for instance specify transformations. Although it is generally easy to compute the value of OCL expressions, doing so in such cases is often not enough.

Consider two variables a and b. A relation between them can be expressed in several ways: 1) $a = f(b)$, 2) $b = g(a)$, 3) $h(a) = i(b)$, or 4) $j(a, b) = true$, where f, g, h, i, and j are functions denoting potentially complex OCL expressions involving their arguments. It is easy to compute the value of a given b from 1), or of b given a from 2). However, computing the value of b given a from 1), of a given b from 2), or of a or b given the other from 3) or 4) can be much more complex.

Moreover, models do change, thus potentially invalidating relations. *Synchronization* corresponds to performing appropriate changes to make relations hold again. It is generally not trivial. If changes always happen on one part (e.g., non-derived features, or source models), then relations may be expressed such as the other part (e.g., derived features, or target models) can be computed easily. If changes can happen on any part (e.g., with changeable derived features, or bidirectional transformations), then there is no way to express relations in OCL such that computation is always easy.

Figure 1 illustrates synchronization of two models M_A and M_B related by relation R (decomposable into model-element-level relations). At some point, M_A evolves into M'_A after some changes (denoted by an arrow labeled a), and relation R may not hold between M'_A and M_B. Synchronizing M_B with M'_A consists in evolving it into M'_B by performing some changes (denoted by an arrow labeled b) such as R holds between M'_A and M'_B. Alternatively, changes denoted by arrow b may happen before those denoted by arrow a, and M_B may be the first model to evolve into M'_B, requiring M_A to be evolved into M'_A.

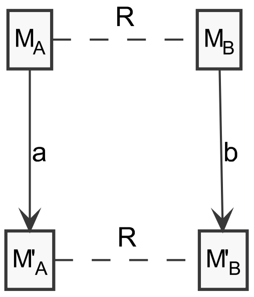

$$\begin{tikzcd}
M_A \arrow[r, dash, dashed, "R"] \arrow[d, "a"] & M_B \arrow[d, "b"] \\
M'_A \arrow[r, dash, dashed, "R"] & M'_B
\end{tikzcd}$$

Fig. 1. Synchronization of models M_A and M_B related by relation R

Active operations [2,1] enable evaluation of operations on collections in a way that is both: 1) incremental (i.e., propagating changes instead of recomputing whole expressions), and 2) bidirectional (i.e., enabling changes to the value of an expression to be propagated back to its source collection). Following a proposal made during the OCL 2014 Workshop panel discussion (see Section 5 of [5]), this paper presents how active operations can be applied to OCL in order to partially address this synchronization problem. A Java implementation of an active operation framework supporting EMF[1] has been developed. As of writing this paper, it is available in a development branch[2] of Papyrus[3]. Manual rewriting of OCL expressions into active operations has been experimented. The resulting active operations have been used in a bidirectional transformation between a profiled UML model, and a model conforming to a metamodel corresponding to the profile. Change propagation in both directions have been extensively tested to behave as expected.

Incrementality is defined in Section 2. Section 3 exposes what active operations consider as immutable and mutable values. Active operations are introduced in Section 4, and their application to OCL is presented in Section 5. In Section 6, our implementation is briefly described. Section 7 discusses some limitations of the approach, along with some ways to mitigate them. Finally, some related works are listed in Section 8, and Section 9 concludes.

[1] Eclipse Modeling Framework: `https://www.eclipse.org/modeling/emf/`

[2] `http://git.eclipse.org/c/papyrus/org.eclipse.papyrus.git/tree/extraplugins/aof?h=committers/fnoyrit/aofacade&id=8bec1ad60253cc854cbd3734efa424bfed0e0bbe`

[3] `https://eclipse.org/papyrus/`

2 Incrementality

When a model is represented by an immutable data structure (e.g., in purely functional approaches), then the only way to perform synchronization is to compute a new model. For derived features, this requires computing a new version of the model in which the corresponding relations hold again. For transformations, the model that was not changed need to be recomputed from the one that did change such that the corresponding relations hold again. However, models are often represented by mutable data structures (e.g., with EMF in Java). In such a case, it is still possible to recompute whole models like with immutable data structures. However, another possibility is to update models in-place by performing small changes that make relations hold again. This is called *incremental* synchronization.

Incremental synchronization has the potential to be more efficient because only relatively small changes are typically required when compared to whole model recomputation. It also updates traceability links between the synchronized models instead of creating new ones with the recomputed model. Moreover, it also avoids creating new elements but rather updates existing ones. This results, for instance, in an interesting advantage when models are being edited in graphical views. Visual shapes are typically bound to the model elements they represent, and are updated when they change. Updating models in-place means that visual editors can directly reflect changes to users. Conversely, recomputation results in whole new models with elements not bound to any visual shape, which means no change is being made visible to users.

Incremental evaluation of OCL expressions can be used to achieve incremental model synchronization. It is based on the same idea of in-place updates, and similarly relies on mutable values[4]. Values resulting from expression evaluation as well as intermediate values corresponding to sub-expressions are updated in-place. For instance, given an ordered set `s` with initial value `OrderedSet {1, 2, 3}`, and expression `s->collect(e | e + 2)->select(e | e > 4)`. The initial evaluation of the expression yields value `OrderedSet {5}` with intermediate value (after the `collect`, but before the `select`): `OrderedSet {3, 4, 5}`. If `s` changes into `OrderedSet {1, 4, 3}` (i.e., `2` is replaced by `4`), then incremental evaluation will start by updating the intermediate value to `OrderedSet {3, 6, 5}` (i.e., replacing `4` by `6`). Then, the value of the whole expression will be updated into `OrderedSet {6, 5}` by adding `6` to it.

3 Mutability of Values

This section starts with values that cannot change before going into mutable values that active operations consider. Finally, it is applied to models.

[4] This does not prevent OCL expressions from always having the same values as if they operated on immutable values, following the OCL specification [9].

3.1 Immutable Values

Primitive values are defined as immutable. These notably include: booleans, numbers, and strings. This is consistent with OCL, and is a choice made by many programming languages (e.g., Java), even if they support mutability.

An immediate consequence is that operations on immutable values do not need to propagate changes. Therefore, these operations are defined in the usual way.

3.2 Mutable Values

Mutable values wrap other (mutable or immutable) values. There are two kinds of mutable values supported by active operations: boxes and objects. A box can be a singleton, or a collection. Each box is *observable*. It notifies *listeners* of its changes: addition, removal, replacement, or move of a wrapped value.

Remark: the value wrapped by a singleton box may be replaced by another one. Since a mutable primitive type variable cannot have its immutable primitive value mutated, it is actually defined as a mutable singleton wrapping a primitive value. In this way, changing the value of the variable actually corresponds to replacing the value wrapped by the singleton box with another one (i.e., changing the content of the box without changing the box itself).

The two kinds of singleton boxes, and four kinds of collection boxes are:

- **Mandatory singletons** (called **one**) are boxes that must contain exactly one value. The contained value can be a different one at different times.
- **Optional singletons** (called **opt**) are boxes that may be empty or contain exactly one value. They may notably become empty, or full.
- **Collections** are boxes that may contain any number of values. They are further classified into four kinds, according to ordering and uniqueness:
  - **Sets** (called **set**) forbid duplicate values and are unordered.
  - **Ordered sets** (called **oset**) forbid duplicate values and are ordered.
  - **Bags** (called **bag**) may contain duplicate values and are unordered.
  - **Sequences** (called **seq**) may contain duplicate values and are ordered.

The type of a box is immutable. This means that, for instance, a **set** will always remain a **set** and never become an **oset**, a **bag**, or a **seq**. The type of elements contained in a box (its *element type*) may be written in brackets (e.g., **one**(String) for a mandatory singleton wrapping a string value). Objects are a special kind of mutable value that contain named slots holding boxes as values.

3.3 Application to Models

Each model element is represented by an object having a fixed type, which is a meta-element coming from a metamodel. The type of an object constrains the types of its slots. Each slot of an object corresponds to a property (attribute or reference) belonging to its meta-element. The type of box used to hold the value of a slot is given by the multiplicity of its corresponding property: an **opt** for 0..1, a **one** for 1..1, and a collection box for n..m with $m > 1$. In the last case, bounds narrower than 0..* are not enforced by the box itself. The specific kind of collection box depends on ordering and uniqueness (as defined above). The element type of a slot value is either a primitive type (for attributes) or a meta-element (for references).

This scheme can be applied to any kind of modeling framework supporting observation, even one not explicitly based on boxes. We have notably applied it to EMF. We leverage the fact that EMF can notify listeners (called `Adapters`) of changes to model element properties to provide a box-based view on models. A box is created to represent each slot, but it delegates storage of its contents to actual `EObjects`: there is no need for content duplication.

4 Active Operations

4.1 Operations on Boxes

Active operations enable bidirectional incremental evaluation of expressions involving boxes. The result of each active operation is also a box. Available operations correspond to well-known OCL operations (e.g., conversion between box types, concatenation, `isEmpty`, `notEmpty`, `size`) and iterators (e.g., `collect`, `select`) on collections or operations available in other languages.

An example of the latter category is the `zipWith` operation (e.g., available in Haskell). `zipWith` operates on two collections, and is given a function (called a `zipper`) taking two arguments. It traverses both collections in parallel, and expects them to have the same size. It returns a collection in which every element is obtained by applying the `zipper` function to one element from each collection. For instance, applying `zipWith` on `OrderedSet {1, 2, 3}` and `OrderedSet {1, 1, 2}` with integer addition as `zipper` function results in `OrderedSet {2, 3, 5}`.

The additional `bind` operation can be used to propagate changes between two result boxes. Each active operation propagates changes from its source(s) to its result, and *vice versa*. It does so using operation-specific algorithms (see [2,1]).

Active operations distinguish between: a) forward change propagation from source to result, and b) reverse change propagation from result to source. Forward direction is always supported, but reverse direction often requires more information. Consequently, reverse direction is only supported if enough information is available. In practice, on a concrete bidirectional transformation, reverse direction can be made to work in most cases. Note that only a) is necessary for incrementality, but both a) and b) are necessary for bidirectional incrementality.

Here is how an operation like `size` works. It observes its source box, and its result is a **one**(Integer) that contains as value the size of the box on which it is applied (its source box). It is updated upon removal or addition of values in its source box. The `size` operation is currently only implemented in a unidirectional way: it only supports forward change propagation. However, limited support for reverse change propagation could make sense, and therefore be implemented. For instance, replacing the value in its result box with a lower value could actually resize its source box by dropping tail elements. Moreover, the size operation could be augmented to a fully bidirectional operation `size(p)` where `p` would be a *provider* function. Provider `p` would return the elements to append to the source box whenever its size increases.

Although this paper is not the place to look at the algorithms between each operation, an overview of how the `collect` operation works is useful. This operation is applied on a source box containing source elements, and is given a *collector* function. It returns a box containing the result of applying the collector to each source element. There are actually ten variants of the `collect` operation[5] exposed to users. Each variant handles a specific combination of change propagation direction (i.e., supporting reverse or not), and mutability of its collector. There is also a forward variant that keeps traceability information, and uses it in order to retrieve already computed elements. Finally, there is a corresponding reverse variant that can read this traceability information.

Reverse change propagation is typically only supported if a reverse collector function is also provided. Property navigation is handled in a special way, and is able to support limited reverse change propagation without requiring a reverse collector. Collecting different values depending on a mutable predicate (e.g., `aCollection->collect(e | if p(e) then f(e) else g(e) endif)`) also requires special handling.

The collector function given to `collect` is either immutable or mutable. An *immutable collector* is a function taking a value as argument, and returning an immutable value. A *mutable collector* is a function taking a value as argument, and returning a box. Collecting with an immutable collector only requires listening for changes on the container boxes (on the source box for forward change propagation, and on the target box for reverse change propagation). Collecting with a mutable collector further requires listening for changes on the result of applying the collector to every source element (called inner boxes).

4.2 Lifting Immutable Operations

Immutable operations (functions or operators) defined on immutable values may be lifted[6] to work on boxes wrapping such immutable values. This enables change propagation for many existing operations. It is trivial for bijective operations such as number or boolean negation, by leveraging `collect`. Non-bijective operations (e.g., absolute value on numbers) can be easily lifted to support forward change propagation on boxes. However, reverse change propagation can generally be performed in several way. For instance, setting the result of an absolute value operation to a positive number (e.g., 5) may be reversed by setting its

[5] Remark: there are also five variants of the `select` operation.

[6] In this context, lifting consists in taking a function operating on simple values, and transforming it into a function operating on boxed values. It works by taking the values out of the boxes before operating on them, and putting them back in boxes afterward.

source to that number (e.g., 5), or to its opposite (e.g., −5). In general, this requires *ad hoc* handling, but a default behavior may be provided (e.g., always returning the positive value). In the general case where no default behavior can solve every problem, users may have to implement the reverse behavior, or at least choose among several possible behaviors (e.g., using annotations).

While `collect` can be used to lift unary operations, binary operations (e.g., the conjunction of boolean values) can be lifted by leveraging `zipWith`. As explained above, there are variants of `collect` without reverse collector that only support forward change propagation, as well as variants with a reverse collector that also support reverse change propagation. When the reverse direction is required to be supported, a reverse collector can be specified to implement default or specific reverse behavior. Similarly, `zipWith` also exists in two variants: one with only a forward zipper function supporting only forward change propagation, and one with an additional reverse zipper function also supporting reverse change propagation. When the reverse direction is required to be supported, a reverse zipper can be specified to implement default or specific reverse behavior.

5 Application to OCL

There are two main aspects to consider in order to apply active operations to OCL: mapping OCL types to active operation types, and rewriting OCL expressions to use active operations such as presented in Section 4.

5.1 Types Mapping

Collection box types and OCL collection types are very closely related, with simple correspondences: **set** for `Set`, **oset** for `OrderedSet`, **bag** for `Bag`, and **seq** for `Sequence`. Model elements are mapped to objects. Although we have not experimented with OCL tuples yet, it seems that they could also map relatively easily to objects.

As for singleton boxes, OCL does not explicitly distinguish nullable values from non-nullable values. However, modeling languages like UML, MOF, and Ecore do. The value of a slot typed by a property with multiplicity [0..1] is mapped to an **opt**. The value of a slot typed by a property with multiplicity [1..1] is mapped to a **one**. In order for every singleton expression to have a definite **one** or **opt** box, static analysis of OCL expressions need to be extended with nullability analysis, which determines whether each sub-expression can actually be null.

5.2 Expression Rewriting

Firstly, operations on primitive values can be made to be the same with active operations and OCL. Secondly, active operations on boxes are quite close to operations on OCL collections. However, several active operations actually correspond to some OCL iterators such as `collect` or `select`. Therefore, in order to know which one to use, static analysis of OCL expressions need also be extended with mutability analysis. That is, whether each sub-expression can actually mutate must be determined. Moreover, complex collector functions need to be rewritten into several simpler functions in order to make sure that each navigation step is actually observed by an active operation. For instance, `persons->collect(p | p.bestFriend.name)` needs to be rewritten into `persons->collect(p | p.bestFriend)->collect(p | p.name)`.

Manually writing relatively complex active operations expressions is cumbersome. But it showed that static mutability analysis can work. Indeed, in our Java-based implementation of active operations, mutable values are distinguished as being instances of interface `IBox`. Any expression that types as an `IBox` is therefore mutable. Of course, this still needs to be implemented for OCL.

Finally, the `bind` active operation generally corresponds to the OCL equality operator applied on collections.

6 Implementation, Debugging and Testing

So far, our implementation of active operations does not handle translation from OCL. However, it does support EMF models, and a significant-enough subset of active operations that enables writing bidirectional incremental transformations.

Debugging is supported by three tools:

- **Inspection.** The first one is the `inspect` pseudo-operation. It has no effect on the data flow, and returns its source box. It is similar to the `trace` function in Haskell, or to the `debug` operation in ATL, except that it stays active and listens to its source box. It logs every change in the console.
- **Data Flow Serialization.** Once an active operations expression has been evaluated, a data flow graph exists in memory to handle change propagation. It is possible to display this data flow graph textually, and to show it in the variable inspection view of the Java debugger.
- **Data Flow Visualization.** Finally, the whole data flow corresponding to the evaluation of all expressions can also be serialized to a textual file. This file can then be further processed by graph layouting tools[7].

Testing active operation expressions (e.g., used for derived features or transformations) has two aspects. First, the passive functionality of the expression (i.e., the computation it initially performs in the absence of change) needs to be tested (e.g., using unit testing). This can be performed using traditional techniques. Second, the active behavior needs to be tested to make sure: 1) that the active operations implementation behaves properly (but this is not the responsibility of a user of active operations), and 2) that the reverse change propagation behaves as intended. The second point is especially important considering that reverse change propagation can often be performed in several ways, but only one may make sense in a given context. We built some tools to do this by comparing the result of change propagation with full passive reexecution.

[7] We use PlantUML: `http://plantuml.com/`.

7 Limitations

Although the approach presented in this paper enables bidirectional incremental evaluation of many useful OCL expressions, it has the following limitations in addition to general bidirectionality issues:

1. Reverse change propagation typically requires more information than available in OCL expressions (e.g., reverse collector or zipper functions).
2. It is not clear yet that all OCL expressions can be rewritten using a fixed set of active operations, and if so, what these operations are.
3. The current active operations algorithms do not support arbitrary-level traversals such as can occur in arbitrary-level collection flattening or closures.

While all these points are open issues, it is possible to mitigate them:

- To mitigate 1:
  - Lifted immutable operations may be annotated with information about which reverse behavior to choose from. For instance, a lifted number addition may be annotated to reverse propagate changes by modifying only one of its operand, or both. If both are modified, it may distribute the change evenly, or not.
  - Specific active operations may be defined with semantics appropriate for a given context. However, this is not trivial since this requires designing a custom bidirectional propagation algorithm.
  - A search-based approach that explores possible solutions to find a "good" one (according to a fitness function) could be used. However, it is unclear at this time how this could interact with active operations algorithms.
  - Finally, if reverse propagation cannot be defined in a meaningful way, it is always possible to make corresponding properties read only in metamodels.
- To overcome 2 and 3, other techniques can be used for forward change propagation (e.g., reevaluating whole sub-expressions). However, reverse change propagation may not be possible in such a case.
- To ease the developer's job despite 1 and 2, a development environment could help. It could warn expression writers of problems (e.g., missing reverse annotation, rewriting impossibility), thus making it easier to write in the appropriate subset of OCL. At least it would make it easy to know when we loose some property (e.g., support for reverse change propagation).

8 Related Work

Firstly, *ad hoc* solutions for bidirectional synchronization can be implemented in general purpose languages such as Java. However, such approaches mix together domain-specific concern with technical aspects. Some frameworks like AngularJS do support data-binding that can perform change propagation. However, the kind of supported relations is quite limited.

Secondly, functional approaches based on lenses can be used to express relations similarly to what can be done with OCL. Some approaches such as [6] enable bidirectional transformations based on lenses. However, functional approaches are not easy to marry efficiently [7] with side-effect based modeling frameworks like EMF.

Finally, there are some graph-pattern based approaches such as IncQuery [10]. IncQuery uses a custom query language based on graph patterns. There is tool support to translate some OCL expressions into these patterns [3]. Although active operations also require some rewriting from OCL, expressions built using active operations are structurally similar to corresponding OCL expressions, whereas graph patterns have different structure. Furthermore, IncQuery may be used as part of VIATRA [4] transformations. VIATRA enables fine-grained custom reactions to specific change events. It should be possible to achieve similar results with active operations, by coupling them with as powerful a transformation language (e.g., possibly by making VIATRA use active operations in addition to IncQuery).

9 Conclusion

This paper proposes an approach that enables incremental (i.e., supporting forward change propagation) evaluation of OCL expressions. Furthermore, this approach can also be made to support reverse change propagation directly in some cases, and with additional annotations in others. Bidirectional incremental evaluation of OCL expressions can thus be achieved in some cases. Applying this approach to the implementation of a model transformation has shown that bidirectionality can be achieved in non-trivial cases.

The approach is based on active operations, of which an overview has been given. Not all OCL expressions can be translated into active operations, and of those which can, not all can propagate changes in the reverse direction. However, it is not clear yet which subset of OCL can be translated, and which (smaller) subset can support reverse propagation. We expect that further works on the rewriting of OCL expressions into active operations will help define these subsets more precisely. The presented approach is still missing automatic rewriting of OCL expressions into active operations, and still has to be evaluated for performance and scalability.

While working on this approach, we noted that it could be useful to have the notion of optional values in OCL. This would enable tools to support null-checks and safe navigation. By combining the approach presented in this paper with the OCLT approach presented in [8], it may become possible to express bidirectional incremental transformations directly in OCL.

References

1. Olivier Beaudoux. *Vers une programmation des systèmes interactifs centrée sur la spécification de modèles exécutables.* Habilitation à diriger des recherches, Université d'Angers, August 2014.

2. Olivier Beaudoux, Arnaud Blouin, Olivier Barais, and Jean-Marc Jézéquel. Active Operations on Collections. In *Model Driven Engineering Languages and Systems*, volume 6394 of *Lecture Notes in Computer Science*, pages 91–105. Springer Berlin Heidelberg, 2010.
3. Gábor Bergmann. Translating OCL to Graph Patterns. In Juergen Dingel, Wolfram Schulte, Isidro Ramos, Silvia Abrahão, and Emilio Insfran, editors, *Model-Driven Engineering Languages and Systems*, volume 8767 of *Lecture Notes in Computer Science*, pages 670–686. Springer International Publishing, 2014.
4. Gábor Bergmann, István Dávid, Ábel Hegedüs, Ákos Horváth, István Ráth, Zoltán Ujhelyi, and Dániel Varró. Viatra 3: A Reactive Model Transformation Platform. In Dimitris Kolovos and Manuel Wimmer, editors, *Theory and Practice of Model Transformations*, volume 9152 of *Lecture Notes in Computer Science*, pages 101–110. Springer International Publishing, 2015.
5. Achim D. Brucker, Tony Clark, Carolina Dania, Geri Georg, Martin Gogolla, Frédéric Jouault, Ernest Teniente, and Burkhart Wolff. Panel Discussion: Proposals for Improving OCL. In *Proceedings of the 14th International Workshop on OCL and Textual Modelling*, pages 83–99, 2014.
6. J Nathan Foster, Alexandre Pilkiewicz, and Benjamin C Pierce. Quotient lenses. In *ACM Sigplan Notices*, volume 43, pages 383–396. ACM, 2008.
7. Sochiro Hidaka and Massimo Tisi. ATLGT: bidirectional ATL on top of GRoundTram. 2015. To appear.
8. Frédéric Jouault, Olivier Beaudoux, Matthias Brun, Mickael Clavreul, and Guillaume Savaton. Towards Functional Model Transformations with OCL. In Dimitris Kolovos and Manuel Wimmer, editors, *Theory and Practice of Model Transformations*, volume 9152 of *Lecture Notes in Computer Science*, pages 111–120. Springer International Publishing, 2015.
9. Object Management Group (OMG). Object Constraint Language (OCL), Version 2.4. `http://www.omg.org/spec/OCL/2.4/`, February 2014.
10. Zoltán Ujhelyi, Gábor Bergmann, Ábel Hegedüs, Ákos Horváth, Benedek Izsó, István Ráth, Zoltán Szatmári, and Dániel Varró. EMF-IncQuery: An integrated development environment for live model queries. *Science of Computer Programming*, 98:80–99, 2015.

Lazy Evaluation for OCL

Massimo Tisi[1], Rémi Douence[2], Dennis Wagelaar[3]

[1] AtlanMod team (Inria, Mines Nantes, LINA), Nantes, France
`massimo.tisi@mines-nantes.fr`
[2] Ascola team (Inria, Mines Nantes, LINA), Nantes, France
`remi.douence@mines-nantes.fr`
[3] HealthConnect NV, Vilvoorde, Belgium
`dennis.wagelaar@healthconnect.be`

Abstract. The Object Constraint Language (OCL) is a central component in modeling and transformation languages such as the Unified Modeling Language (UML), the Meta Object Facility (MOF), and Query View Transformation (QVT). OCL is standardized as a strict functional language. In this article, we propose a lazy evaluation strategy for OCL. We argue that a lazy evaluation semantics is beneficial in some model-driven engineering scenarios for: i) lowering evaluation times on very large models; ii) simplifying expressions on models by using infinite data structures (*e.g.*, infinite models); iii) increasing the reusability of OCL libraries. We implement the approach on the ATL virtual machine EMFTVM.

1 Introduction

The Object Constraint Language (OCL) [1] is widely used in model-driven engineering (MDE) for a number of different purposes. For instance, in the Unified Modeling Language (UML), OCL expressions are used to specify: queries, invariants on classes and types in the class model, type invariants for stereotypes, pre- and post-conditions on operations and methods, target (sets) for messages and actions, constraints on operations, derivation rules for attributes. Besides its role in UML, OCL is embedded as expression language within several MDE languages, including metamodeling languages (e.g., the Meta Object Facility, MOF) and transformation languages (e.g., the Query View Transformation language, QVT, and the AtlanMod Transformation Language, ATL [2]).

In the standard specification of the OCL semantics [1], the language is defined as a side-effect-free functional language. While several implementations of the specification exist as a standalone language (e.g., [3]), or as an embedded expression language (e.g., in [2]), they all compute OCL expressions by a strict evaluation strategy, *i.e.*, an expression is evaluated as soon as it is bound to a variable. Conversely, a lazy evaluation strategy, or call-by-need [4] would delay the evaluation of an expression until its value is needed, if ever. In this paper we want to: 1) clarify the motivation for lazy OCL evaluation and capture the main opportunities of application by means of examples; 2) propose a lazy evaluation strategy for OCL by focusing on the specificities of the OCL language *w.r.t.* other functional languages; 3) present an implementation of the approach in the ATL virtual machine EMFTVM[4] [5].

The first effect we want to achieve is a performance increase in some scenarios by avoiding needless calculations. Companies that use MDE in their software engineering processes need to handle large amounts of data. In MDE, these data structures would translate into very large models (VLMs), *e.g.*, models made by millions of model elements. Examples of such model sizes appear in a range of domains as shown by industrial cases from literature: AUTOSAR models [6], civil-engineering models [7], product families [8], reverse-engineered software models [9]. A lazy evaluation strategy for a model navigation language like OCL would allow to 1) delay the access to source model elements to the moment in which this access is needed by the application logic and, by consequence, 2) reduce the number of processed model elements, by skipping the unnecessary ones (if any). When the OCL evaluator is embedded in an MDE tool, lazy OCL evaluation may have a significant impact on the global tool performance.

Our second purpose is enabling the use of infinite data structures in the definition of algorithms with OCL. Indeed, infinite data structures make some algorithms simpler to program. For instance, they allow to decouple code in a producer-consumer pattern: a producer function defines data production without caring for the actual quantity of data produced; a consumer function explores the data structure, implicitly driving the production of the necessary amount of data. For instance, it is simpler to lazily generate infinite game trees and then explore them (e.g., by a min-max algorithm), rather than estimating at each move the part of the game tree to generate. In this paper we argue that infinite data structures simplify also the development of common queries in MDE.

Finally our third objective is to use laziness to improve the reusability of OCL libraries, by reducing their dependencies. Indeed, laziness promotes definitions reuse. For instance, the minimum of a collection can be defined as the composition of sorting with selection of the first element. Such a definition reuses code but it can be vey inefficient in a strict evaluation strategy, requiring the full collection sorting. Laziness makes it practical, at least for some sorting algorithms, since only the computation for sorting the first element will be executed. Similarly, composing libraries in a producer-consumer pattern, enables the definition of general (hence reusable) generators that compute many (possibly infinite) results. Consumers specialize generators to the context of use by demanding only part of the generated elements.

The remainder of this paper is organized as follows: Section 2 motivating the need for lazy evaluation in OCL by introducing two running scenarios; Section 3 describes our approach; Section 4 discusses the implementation strategy; Section 5 lists the main related works; Section 6 concludes the paper with a future research plan.

[4] available from `http://wiki.eclipse.org/ATL/EMFTVM`

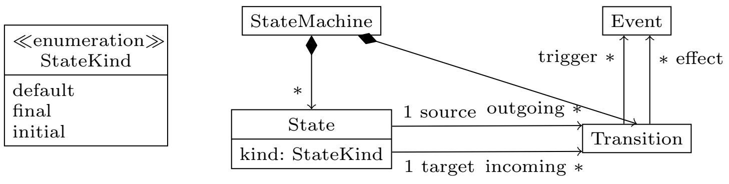


Fig. 1. State Machine metamodel (excerpt)

2 Motivating Examples

This section introduces two examples of OCL queries with the purpose of highlighting the benefits of lazy evaluation in the specific case of model queries.

The state machine in Fig. 2 conforms to the State Machine metamodel displayed in Fig. 1. This metamodel defines a `StateMachine` as composed of several `State` elements. A *kind* property is used to distinguish special states, such as the unique `initial` state and possibly several `final` states. `Transitions` connect couples of `States`. Each transition is triggered by a set of events (`trigger`) and when it fires it produces new events (`effect`).

We provide this state machine with a simple execution semantics. The machine maintains a queue of events to process, that is initially not empty. The execution starts from the initial state and checks the top of the queue for events that match the trigger of some outgoing transition. If such events are found, the transition is fired: the machine moves to the target state of the transition, the triggering events are removed from the top of the queue and the effect events are added to the bottom of the queue. In our simple model the machine proceeds autonomously (no external events are considered) and deterministically (triggers outgoing from the same state are disjoint).

2.1 Queries on Large Models

As a first example scenario we check if there exists a non-final state that contains a self-transition[5]:

[5] The query structure is identical to the one introduced in [9] and used in several works to compare the execution performance of query languages, but here we apply it to state machines instead of class diagrams.

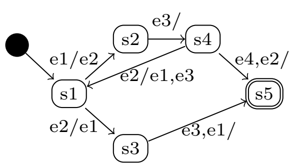


Fig. 2. State machine example (transitions are labeled as *trigger/effect*)

```
State.allInstances()->select(s | not s.kind = 'final')
  ->exists(s | s.outgoing->exists(t | t.target = s))
```

If we assume the number of states of the input state machine to be very large, the time and memory cost to evaluate such query may be high. Here are the steps that a strict evaluation of OCL typically performs:

1. **Computation of the extent of class `State`.** In this first step, the OCL evaluator typically traverses the whole model on which the query is evaluated in order to compute the collection of all elements that have `State` as type (directly, or indirectly via inheritance).
2. **Filtering out final states.** Then, the whole collection computed in previous step is traversed in order to keep only states that are not final.
3. **Finding a state with a self-transition.** Finally, the list of non-final states is traversed in order to discover if one of them satisfies the condition.

Several optimizations may be supported by an OCL evaluator. For instance, with **extent caching**, the result of calling `allInstances()` on a model for a given type (`State` in our example) may be cached. Thus, a second extent computation will not require traversal of the whole model. In our case, this will not reduce the cost of the first evaluation, but will reduce the cost of subsequent evaluations (provided the source model is not modified, which may invalidate our cache, and require a new extent computation). However, with these optimizations alone, even if a non-final state satisfying the condition appears near the beginning of the model, the whole model still needs to be traversed for the first computation of the extent of `State`, and the whole list of states needs to be traversed for each evaluation in order to filter out final states.

Especially when the query is performed as part of an interactive tool, there may be a significant need to reduce the query response time. Moreover, if the queried model is too large to fit in RAM (e.g., it may be stored in a database and traversed lazily using frameworks such as CDO[6]), evaluation of the query will simply fail. In such a case, the computation of the extent of `Class` will force all elements typed by `Class` to be loaded into RAM (at least a proxy per element if not the values of all their properties). However, we do not actually need all such elements to be in memory at the same time.

2.2 Infinite Collections in Model Queries

In the queries of this section we consider also the state machine semantics and in particular event consumption. The following OCL query computes if a final state is reachable from the current state in a given number of steps while consuming all the given events (in this case we say that the state is *valid*):

```
context State::isValid (events:Sequence(Event), steps:Integer) : Boolean
body :
  if (steps<0) then false else
    if (events->isEmpty()) then self.kind = 'final'
```

[6] `http://www.eclipse.org/cdo/`

```
    else self.outgoing->exists(t | events->startsWith(t.trigger)[7]
      and t.target.isValid(events->difference(t.trigger)
        ->union(t.effect)), steps-1);
    endif
  endif;
```

The following query searches for a repeating state in the state machine execution (e.g., to possibly optimize the state machine execution):

```
context State :: repeatingState (events: Sequence(Event)) : State body :
  self.repeatingStateRec (events, Set{});
context State :: repeatingStateRec (events: Sequence(Event),
  visited: Set(State)) : State body :
  if (visited->includes(self)) then self
  else self.outgoing->select(t | events->startsWith(t.trigger))
    ->any().target.repeatingStateRec(events->difference(t.trigger)
      ->union(t.effect), visited->including(self)))
  endif;
```

The logic of the two recursive queries have clear similarities, being both based on a simulation of the state-machine execution. However the simulation logic is embedded in the query definitions, and interleaved with query-specific logic, i.e. validity or repetition checks. Factorizing the logic for state-machine simulation would simplify the definition of the queries, avoid code duplication, and increase code-reusability. We may try to achieve this factorization by writing a `simulate` OCL query that given a set of events returns an execution trace:

```
context State :: simulate (events: Sequence(Event)) :
  Sequence(Tuple(state: State, events: Sequence(Event))) body :
  let tr : Transition = self.outgoing
    ->select(t | events->startsWith(t.trigger))->any() in
  Sequence{Tuple{state=self, events=events}}
    ->union(tr.target.simulate(events->difference(tr.trigger)
        ->union(tr.effect));
```

Reusing the `simulate` function considerably simplifies the definition of the previous queries, that can be re-written as:

```
context State :: isValid(events:Sequence(Event), steps:Integer): Boolean
body : self.simulate(events)->subSequence(1,steps)
    ->exists(tu | tu.state.kind = 'final' and tu.events->isEmpty());
```

```
context State :: repeatingState (events: Sequence(Event)) : Boolean
body : self.simulate(events)->collect(tu | tu.state)->firstRepeating()[8];
```

However the result of `simulate` is in general an infinite sequence of states and the use we describe would be possible only by providing OCL with a lazy semantics.

3 Lazy Evaluation of OCL

3.1 Approach Overview

In general, lazy evaluation consists in delaying computations, detecting when the result of such a delayed computation is needed, and forcing the delayed compu-

[7] `startsWith` is a shortcut for as `self.subSequence(1,argument->size())=argument`

[8] `firstRepeating` is defined as an operation on ordered collections (independent from state machines) finding the first repeating occurrence

tation. In functional languages, there is a single way to define computation: functions. When function (application) is lazy, the language is lazy. Object-oriented languages (with late binding) do not fit well with laziness. Indeed, evaluation of a method call requires to evaluate its receiver in order to lookup the method definition. Overloading also requires to evaluate arguments. Hence, method call in object orientation is essentially strict. For this reason, in OCL we choose to restrict laziness to collections. Our approach relies on iterators which allow us to produce and consume incrementally (lazily) the elements of a collection.

3.2 Laziness and the OCL Specification

One of the main design goals of our approach for lazy OCL is maximizing compatibility with standard (strict) OCL.

We choose not to extend or change the OCL syntax. In particular we avoid introducing language constructs to control if an expression (or data value, function call...) will be eagerly/lazily computed, like `strict/lazy` keywords or explicit lazy data types (e.g., `LazySet`). This enables programmers to directly reuse existing programs and libraries. We also argue that this choice preserves the advantage of declarative languages like OCL, i.e. programmers do not need to worry about how statements are evaluated. As we will see in the next section, keeping laziness completely implicit is indeed a challenge for the lazy evaluation of high-level declarative languages like OCL.

We also do not change the semantics of existing terminating OCL programs: if a query terminates in strict OCL and returns a value, it also terminates in lazy OCL and returns the same value (although it may require less computation to do so). The only exception to this property are queries that during their computation produce an `invalid` value, as we will soon see.

We are not only backward compatible, but some non-terminating OCL queries terminate in lazy OCL. In particular, we allow the definition of infinite collections and the application of OCL collection operations to them, with some restrictions that we discuss in the next section. Queries that make use of infinite collections terminate, as long as only a finite part of the collection is required by the computation. This is a deviation (extension) of the OCL standard, which defines that all collections are finite: potential infinite sets such as `Integer.allInstances()` are `invalid` in the standard.

As we mentioned, the error management mechanism of OCL has a significant impact on the backward compatibility of the lazy semantics. In OCL, errors are represented as `invalid` values that propagate: for instance when `invalid` is added to a collection, the resulting whole collection is `invalid`. In lazy OCL, the value of an element is unknown until it is accessed. So, if an `invalid` element is never accessed, it does not propagate and the prefix of the collection is well defined. This means that strict queries that return `invalid`, may return a different value in lazy semantics.

Moreover OCL provides the programmer with the `oclIsInvalid` function to handle `invalid` values (somehow analogously to catching exceptions in Java). The function returns `true` if its argument is `invalid` and at the same time stops the propagation of `invalid`, allowing the program to recover from the error and possibly terminate correctly. Hence terminating queries in strict semantics that use `oclIsInvalid` may produce a different valid value than the same queries in lazy semantics.

Summarizing: 1) expressions that return a valid result in strict semantics return the same result in lazy semantics, 2) expressions that return an invalid result or do not terminate in strict semantics may return a valid result in lazy semantics, 3) expressions that use the `oclIsInvalid` function are an exception to (1) and (2), as they are in general not compatible with the lazy semantics. Note that the other special OCL value, `null`, is a valid value that can be owned by collections, hence it does not pose any compatibility problem to the lazy semantics.

3.3 OCL Operations

OCL functions benefit from laziness in a different degree. In Table 1 we list all the OCL operations on collections and Table 2 all the iterators (according to [1]). For each operation, and each kind of collection it can be applied to, we provide two properties that characterize its lazy behavior:

– We add a constraint to the *Restrictions* column to indicate that the operation/iterator may not terminate, or it is simply not well-defined, if its source (context) or argument is an infinite collection. Examples of such cases are: appending an element at the end of an infinite collection, reversing it, calculating its maximum.
– We specify in the *Strictness* column if the operation/iterator always evaluates the totality of the source or argument collection. Simple examples are: sorting the collection, summing it, or generically iterating over it (`iterate`).

The properties in Tables 1 and 2 implicitly categorize OCL operations and iterators w.r.t. laziness: operations that can be lazily applied without restrictions (e.g., `product`), operations that can lazily navigate only some of the arguments (e.g., `src - c` lazily navigates the source/context collection `src` but strictly evaluates the argument `c`) and operations that do not support lazy evaluation (e.g., `iterate`).

For brevity, in the following we illustrate in detail only a subset of the OCL functions. The reader may extend the principles we introduce to analogous functions.

AllInstances. While not being an operation in the context of a collection type, `allInstances` returns a collection, made by the instances of the type in argument, and this collection can be lazily computed. OCL implementations usually perform a depth-first traversal on the model containment tree to find the model instances and populate the result collection, but this traversal order is not defined in the OCL specification. We propose a lazy evaluation semantics for `allInstances` that supports the navigation of infinite models. However, even

Table 1. Laziness for OCL collection operations

Context	Operation	Restrictions	Strictness
Collection	`=/<> (c : Collection)`	`src` and `c` finite	-
Collection	`size ()`	`src` finite	strict on `src`
Collection	`includes/excludes (o : OclAny)`	`src` finite	-
Collection	`includesAll/excludesAll (c : Collection)`	`src` and `c` finite	-
Collection	`isEmpty/notEmpty ()`	-	-
Collection	`max/min/sum ()`	`src` finite	strict on `src`
Set/Bag	`including (o : OclAny)`	-	-
OrdSet/Sequence		`src` finite	-
Collection	`excluding (o : OclAny)`	-	-
Set/Bag	`union (c : Collection)`	-	-
OrdSet/Sequence		`src` finite	-
Collection	`product (c : Collection)`	-	-
Collection	`selectByKind/selectByType (t: OclType)`	-	-
Collection	`asSet/asOrdSet/asSequence/asBag ()`	-	-
Set/Bag	`flatten (c : Collection)`	-	-
OrdSet/Sequence		`src` finite	-
Set/Bag	`intersection (c : Collection)`	-	-
Set	`- (c : Set)`	`c` finite	strict on `c`
Set	`symmetricDifference (c : Set)`	`src` and `c` finite,	strict on `src` and `c`
OrdSet/Sequence	`append (o : OclAny)`	`src` finite	-
OrdSet/Sequence	`prepend (o : OclAny)`	-	-
OrdSet/Sequence	`insertAt (n : Integer, o : OclAny)`	-	-
OrdSet/Sequence	`subOrdSet/subSeq (f : Integer, l : Integer)`	-	-
OrdSet/Sequence	`at (n : Integer)`	-	-
OrdSet/Sequence	`indexOf (o : OclAny)`	-	-
OrdSet/Sequence	`first ()`	-	-
OrdSet/Sequence	`last ()`	`src` finite	-
OrdSet/Sequence	`reverse ()`	`src` finite	-

Table 2. Laziness for OCL collection iterators

Iterator	Restrictions	Strictness
`iterate`	`src` finite	strict on `src`
`any`	-	-
`closure`	`src` finite	strict on `src`
`collect`	-	-
`collectNested`	-	-
`count`	`src` finite	strict on `src`
`exists`	`src` finite	-
`forAll`	`src` finite	-
`isUnique`	`src` finite	-
`one`	`src` finite	-
`reject`	-	-
`select`	-	-
`sortedBy`	`src` finite	strict on `src`

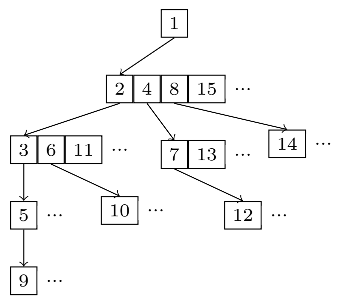


Fig. 3. Fair-first traversal (numbers indicate traversal order)

when models are finite but large, lazy `allInstances` can lead to easier programming and better performance.

Applying `allInstances` to infinite models is not trivial. Depth-first traversal of the containment tree does not work in the case of models with infinite depth: a query like `Type.allInstances()->includes(e)` will not terminate if `e` appears in a rightmost branch w.r.t. an infinite-depth branch. Dually a breadth-first traversal will not work if the model contains a node with infinite children: the traversal will never move to the next tree level. According to our principle of *implicit* laziness, we avoid introducing user-defined model traversals (e.g., `State.allInstancesBreadthFirst()`), that would leave to the user the burden of selecting the correct traversal strategy for `allInstances` depending on the model structure.

Instead, we propose a specific model-traversal order for lazy evaluation of `allInstances`. We still traverse the containment tree with a traversal strategy that alternates at each step a movement in depth and one in width (in an ideally *diagonal* way). Listing 1.1 formalizes the semantics of the traversal in Haskell (function `fairFS`) and Figure 3 graphically illustrates the traversal order.

For instance, applying the example query of Section 2.1 in strict semantics to a state machine with infinite states, the first `allInstances` would never terminate and the following select would never start computing. In our lazy semantics instead, `allInstances` would traverse the infinite model by need, and the full query would actually terminate if a non-final state containing a self-transition was found.

Listing 1.1. Fair traversal for lazy semantics of `allInstances`

```
class Tree t where
    subs :: t a -> [t a]

data RoseTree a = Node a [RoseTree a] deriving Show

label :: RoseTree a -> a
label (Node l _) = l

instance Tree RoseTree where
    subs (Node _ ts) = ts

```

```
type Visitor a = a -> [a]

type Brothers t = [t]

nextBrother :: Visitor (Brothers t)
nextBrother [_]    = []
nextBrother (_:ts) = [ts]

nextSon :: Tree t => Visitor (Brothers (t a))
nextSon (t:_) | null (subs t) = []
              | otherwise     = [subs t]

fairFS :: Tree t => Visitor (t a)
fairFS t = ffsIter [[t]]
   where ffsIter (ts:tss) = head ts:ffsIter (tss++nextSon ts++nextBrother ts)
         ffsIter [] = []
```

Union. The `union` operator computes the union of two collections. In lazy OCL each collection is represented as an iterator of elements, hence their `union` is also represented as an iterator of elements.

Four versions of the union, in function of the type of their arguments, are detailed in Listing 1.2. When the collection arguments are `Sequences` the union appends (recursively) the elements of the first collection to the head of the second one. When the arguments are `Bags` the union is a fair interleaving of the two collections. When the arguments are `OrderedSets` the union concatenates the first collection to the second, but elements of the first collection are deleted from the second, to preserve the unicity property. Finally, when the arguments are `Sets`, the union interleaves the two collections while deleting duplicated elements.

The different lazy behavior of the four `union` semantics stands out when they are used with infinite collections. When collections are not ordered no restriction is required, since the interleaving allows to fairly navigate and merge both of the infinite collections. When the collections are ordered if the first argument is infinite the elements of the second arguments will not occur in the infinite result, because in the declarative semantics of OCL the elements of the first collection must occur before the elements of the second collection. In other words, if `c1` is infinite and ordered, than `c1.union(c2)` is equivalent to `c1` for all uses in OCL.

Listing 1.2. Lazy `union`

```
unionSequence (x:xs) ys = x:unionSequence xs ys
unionSequence []     ys = ys

unionOrderedSet (x:xs) ys = x:unionOrderedSet xs (delete x ys)
unionOrderedSet []     ys = ys

unionBag (x:xs) ys = x:unionBag ys xs
unionBag []     ys = ys

unionSet (x:xs) ys = x:unionSet (delete x ys) xs
unionSet []     ys = ys
```

Intersection. The *intersection* operator computes the intersection of two Sets or Bags. Such a computation requires an occurrence check (an element belongs to the result only if it belongs to both collections), that is in general an operation that is not applicable to infinite sets.

In the lazy execution algorithm we propose (Listing 1.3) both collections are inspected in parallel (note how the arguments are swapped in the recursive call) and the check of occurrence in the other collection is performed with respect to the already considered elements.

Listing 1.3. Lazy `intersection`

```
intersect xs ys = intersect' xs [] ys []
    where intersect' (x:xs) seenInXs ys seenInYs
              | x `elem` seenInYs = x:intersect' ys (delete x seenInYs) xs
                  seenInXs
              | otherwise         = intersect' ys seenInYs xs seenInXs
          intersect' [] _ _ _ = []
```

Table 3 shows an example of execution trace of this algorithm where two infinite integer collections are intersected. In the columns of Table 3 we show, for each step: the part of the collections that is still to evaluate (columns `set1` and `set2`), the elements of the two collections that have already been considered (columns `buffer1` and `buffer2`), the test applied at the current step (column `test`), the result being built (column `set1∩set2`).

Table 3. Example of lazy intersection: {powers of 2} ∩ {squares}

set1 (powers of 2)	buffer1	test	buffer2	set2 (squares)	set1∩set2
{1,2,4,8,16,32,64,128,256...}	{}		{}	{1,4,9,16,25,36,49,64...}	{}
{2,4,8,16,32,64,128,256...}	{}	$1 \notin$	{}	{1,4,9,16,25,36,49,64...}	{}
{2,4,8,16,32,64,128,256...}	{1}	$\ni 1$	{}	{4,9,16,25,36,49,64...}	{}
{4,8,16,32,64,128,256...}	{}	$2 \notin$	{}	{4,9,16,25,36,49,64...}	{1}
{4,8,16,32,64,128,256...}	{2}	$\not\ni 4$	{}	{9,16,25,36,49,64...}	{1}
{8,16,32,64,128,256...}	{2}	$4 \in$	{4}	{9,16,25,36,49,64...}	{1}
{8,16,32,64,128,256...}	{2}	$\not\ni 9$	{}	{16,25,36,49,64...}	{1,4}
{16,32,64,128,256...}	{2}	$8 \notin$	{9}	{16,25,36,49,64...}	{1,4}
{16,32,64,128,256...}	{2,8}	$\not\ni 16$	{9}	{25,36,49,64...}	{1,4}
{32,64,128,256...}	{2,8}	$16 \in$	{9,16}	{25,36,49,64...}	{1,4}
{32,64,128,256...}	{2,8}	$\not\ni 25$	{9}	{36,49,64...}	{1,4,16}
{64,128,256...}	{2,8}	$32 \notin$	{9,25}	{36,49,64...}	{1,4,16}
{64,128,256...}	{2,8,32}	$\not\ni 36$	{9,25}	{49,64...}	{1,4,16}
{128,256...}	{2,8,32}	$64 \notin$	{9,25,36}	{49,64...}	{1,4,16}
{128,256...}	{2,8,32,64}	$\not\ni 49$	{9,25,36}	{64...}	{1,4,16}
{256...}	{2,8,32,64}	$128 \notin$	{9,25,36,49}	{64...}	{1,4,16}
{256...}	{2,8,32,64,128}	$\ni 64$	{9,25,36,49}	{...}	{1,4,16}
{...}	{2,8,32,64,128}	$256 \notin$	{9,25,36,49}	{...}	{1,4,16,64}

4 Lazy OCL in ATL/EMFTVM

We have implemented lazy OCL evaluation upon the ATL virtual machine EMFTVM. We compile the underlying OCL expression into imperative byte codes, like INVOKE, ALLINST and ITERATE as explained in [5]. In order to lazily evaluate collections, we implemented the `LazyCollection` type and its subtypes, `LazyList`, `LazySet`, `LazyBag`, and `LazyOrderedSet` corresponding to four collection types of OCL (`Sequence`, `Set`, `Bag` and `OrderedSet`). An iterator such as `select` or `collect` does not immediately iterate over its source collection, but rather returns a lazy collection to its parent expression that keeps a reference to the source collection, and to the body of the iterator. This is possible because EMFTVM supports closures (also known as lambda-expressions). Then, when a collection returned by an iterator is traversed, it only executes the body of the iterator on the source elements as required by the parent expression.

Listing 1.4 for instance shows the relevant code excerpts for implementing the `collect` operation for `Bags`. A `LazyBag` class extends `LazyCollection` and defines methods for each operation on `Bags`, e.g. `collect()`. In the strict version the `collect()` method would contain the code for computing the resulting collection (i.e., applying the argument function to each element of the source collection). In our lazy implementation the method just returns another `LazyBag`. A `LazyBag` is constructed by passing an `Iterable` as the data source of the collection. In the case of `collect` the `Iterable` is built around a `CollectIterator` (from `LazyCollection`), and the `collect` logic is embedded in the two methods `next()` and `hasNext()` of the iterator. In the `CollectIterator` the `next()` method executes a `function CodeBlock`, representing the lamba-expression associated with it.

Listing 1.4. LazyCollection

```
public class LazyBag<E> extends LazyCollection<E> {
    // ...
    /**
     * Collects the return values of <code>function</code> for
     * each of the elements of this collection.
     * @param function the return value function
     * @return a new lazy bag with the <code>function</code> return values.
     * @param <T> the element type
     */
    public <T> LazyBag<T> collect(final CodeBlock function) {
        // ...
        return new LazyBag<T>(new Iterable<T>() {
            public Iterator<T> iterator() {
                return new CollectIterator<T>(inner, function, parentFrame);
            }
        });
    }
    // ...
}
public abstract class LazyCollection<E> implements Collection<E> {
    //...
    public static class CollectIterator<T> extends ReadOnlyIterator<T> {

        protected final Iterator<?> inner;
        protected final CodeBlock function;
        protected final StackFrame parentFrame;

        /**
         * Creates a {@link CollectIterator} with <code>condition</code> on <
             code>inner</code>.
         * @param inner the underlying collection
         * @param function the value function
         * @param parentFrame the parent stack frame context
         */
        public CollectIterator(final Iterable<?> inner, final CodeBlock
            function, final StackFrame parentFrame) {
            super();
            this.inner = inner.iterator();
            this.function = function;
            this.parentFrame = parentFrame;
        }

        public boolean hasNext() {
            return inner.hasNext();
        }

        public T next() {
            return (T) function.execute(parentFrame.getSubFrame(function,
                 inner.next()));
        }
    }
    // ...
}
```

In EMFTVM, `allInstances()` returns a lazy list that traverses the source model lazily, as illustrated in Listing 1.5. The method `allInstancesOf()` in the class `ModelImpl` is executed at each call to OCL `allInstances`. The method returns a `LazyList` whose data source is a `ResourceIterable`. `ResourceIterable` contains a `DiagonalResourceIterator` that implements in its `next()` method the fair tree traversal strategy specified in Listing 1.1[9].

Listing 1.5. allInstances

```
public class ModelImpl extends EObjectImpl implements Model {
    // ...
    public LazyList<EObject> allInstancesOf(final EClass type) {
        return new LazyList(new ResourceIterable(getResource()), type));
    }
    // ...
}
public class ResourceIterable implements Iterable<EObject> {
    // ...
    public Iterator<EObject> iterator() {
        // the DiagonalResourceIterator implements the fair tree traversal
        return new DiagonalResourceIterator<EObject>(this, false)
    }
    // ...
}
```

Our implementation allows to define and use lazy queries on very large or infinite models, including the examples of Section 2. We have not performed a systematic performance experimentation and time execution performance of the lazy implementation clearly depends on the ratio of the large collections that is actually visited by the query. When performance is the main concern, lazy semantics has to be preferred if a small part of collections is used; strict semantics is still faster in other cases because of the lower overhead.

As an example, we perform the OCL query from Section 2.1 in a strict way with the classic (strict) ATL virtual machine and in a lazy way with the lazy

[9] Note that the current implementation of `allInstances()` in standard ATL returns a `Sequence` of elements in depth-first order, instead of a `Set`. This deviation from the OCL standard may improve the engine performance (by avoiding occurrence checks). The drawback is that the traversal order is exposed to the user, that can consider it in its transformation. In such cases our change in traversal order may break backward-compatibility.

EMFTVM on an Intel core i7, 2.70GHz x 8, x86_64 CPU with 8GiB of RAM. We provide a large state machine made of 38414 elements, where the first state satisfies the query condition. Then, we compare results returned from the two OCL evaluation methods and summarize them in Table 4. The column `Calls` presents the number of operation calls on elements of the underlying collection, i.e., iterations over the `->select()` and the `->exists()`. As shown in Table 4, the lazy evaluator stops the iteration on both `->select()` and `->exists()` as soon as the condition is satisfied (i.e., for the first state), resulting in a much faster execution.

5 Related Work

Lazy evaluation of functional languages is a subject with a long tradition [4], yet it is still studied [10]. We refer the reader to [11] for an example based on Lisp, and to [12] for its formal treatment. Lazy evaluation can be mixed with strict one [13][14]. Hughes has argued that laziness makes programs more reusable [15]. Our approach based on lazy iterators is a simplified version of *iteratees* [16]. Indeed, iteratees are composable abstractions for incrementally processing of sequences. However, our iterators do not isolate effects with a monad, nor distinguish producers, consumers and transducers. Moreover, in our iterators either there is a next value or the iteration is over, but we do not consider raising errors.

The idea of defining and using infinite models has been already addressed in previous work. In [17] transformation rules are lazily executed, producing a target model that can be in principle infinite. In [18] the authors extend MOF to support infinite multiplicity and study co-recursion over infinite model structures. Both works do not provide the query language with an explicit support of infinity. Streaming models can be considered a special kind of infinite models, and their transformation has been recently studied in [19] with languages like IncQuery, but the focus is more on incrementality than laziness.

As alternatives to laziness, other improvements to OCL evaluation have been explored in several works. In [20] the OCL execution engine has been optimized "locally" (i.e., by changing code generated for a given construct). With laziness, we perform only the necessary iterations in many more cases. However,

Table 4. Lazy vs. Strict OCL evaluation in ATL.

Query	**Model Size**	**Lazy Eval.**		**Strict Eval.**	
		Calls	Time	Calls	Time
Example 1	38414	2	0.002 s	38412	0.200 s
`State.allInstances()->select(s | not s.kind = 'final')`		1		25608	
`->exists(s | s.outgoing->exists(t | t.target = s))))`		1		12804	

from a performance point of view, laziness overhead should also be considered. The paper in [21] proposes a mathematical formalism that describes how the implementation of standard operations on collections can be made active. In that way they could evaluate the worst case complexities of active loop rules on collections with a case study. The work in [22] reports on the experience developing an evaluator in Java for efficient OCL evaluation. They aim to cope the novel usages of the language and to improve the efficiency of the evaluator on medium-large scenarios. [23] proposes to extend the OCL evaluator to support immutable collections. Finally, an issue tightly coupled to lazy navigation, is on-demand physical access to the source model elements, i.e. lazy loading. For lazy loading of models for transformation we refer the reader to [24].

6 Conclusions

In this paper we argue that a lazy evaluation semantics for OCL expressions would increase the performance of OCL evaluators in some scenarios, simplify the definition of some queries and foster the development of more reusable OCL libraries in a producer-consumer pattern. We illustrates by example the main challenges of lazy OCL, we provide novel lazy algorithms for some OCL operations (i.e., `allInstances` and `intersection`) and perform an implementation of the approach in the ATL virtual machine EMFTVM.

In future work we plan to perform an extensive performance evaluation on a corpus of real-world OCL queries used in ATL transformation projects. From this study we plan to derive a systematic approach for identifying queries that benefit from lazy evaluation.

Acknowledgements. This work was partially funded by the AutoMobile EU 7th FP SME Research project.

References

1. OMG: Object Constraint Language Specification, version 2.4. Object Management Group. (February 2014)
2. Jouault, F., Allilaire, F., Bézivin, J., Kurtev, I.: Atl: A model transformation tool. Sci. Comput. Program. **72**(1-2) (2008) 31–39
3. Eclipse Model Development Tools Project: Eclipse OCL website `http://www.eclipse.org/modeling/mdt/?project=ocl`.
4. Wadsworth, C.P.: Semantics And Pragmatics Of The Lambda-Calculus. PhD thesis, University of Oxford (1971)
5. Wagelaar, D., Tisi, M., Cabot, J., Jouault, F.: Towards a general composition semantics for rule-based model transformation. In: MoDELS. (2011) 623–637
6. Bergmann, G., Horváth, Á., Ráth, I., Varró, D., Balogh, A., Balogh, Z., Ökrös, A.: Incremental evaluation of model queries over emf models. In: Model Driven Engineering Languages and Systems. Springer (2010) 76–90
7. Steel, J., Drogemuller, R., Toth, B.: Model interoperability in building information modelling. Software & Systems Modeling **11**(1) (2012) 99–109

8. Pohjonen, R., Tolvanen, J.P., Consulting, M.: Automated production of family members: Lessons learned. In: Proceedings of the Second International Workshop on Product Line Engineering-The Early Steps: Planning, Modeling, and Managing (PLEES'02), Citeseer (2002) 49–57
9. Sottet, J.S., Jouault, F.: Program Comprehension. GraBaTs 2009 Case Study, `http://www.emn.fr/z-info/atlanmod/index.php/GraBaTs_2009_Case_Study`
10. Chang, S., Felleisen, M.: Profiling for laziness. SIGPLAN Not. **49**(1) (January 2014) 349–360
11. Henderson, P., Morris, Jr., J.H.: A lazy evaluator. In: Proceedings of the 3rd ACM SIGACT-SIGPLAN symposium on Principles on programming languages. POPL '76, ACM (1976) 95–103
12. Ariola, Z.M., Maraist, J., Odersky, M., Felleisen, M., Wadler, P.: A call-by-need lambda calculus. In: Proceedings of the 22Nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. POPL '95, New York, NY, USA, ACM (1995) 233–246
13. Wadler, P., Taha, W., MacQueen, D.: How to add laziness to a strict language, without even being odd. In: Workshop on Standard ML. (1998)
14. Mauny, M.: Integrating lazy evaluation in strict ML. Research Report RT-0137 (1992)
15. Hughes, J.: Why Functional Programming Matters. Computer Journal **32**(2) (1989) 98–107
16. Kiselyov, O., Peyton-Jones, S., Sabry, A.: Lazy v. yield: Incremental, linear pretty-printing. In Jhala, R., Igarashi, A., eds.: Programming Languages and Systems. Volume 7705 of Lecture Notes in Computer Science. Springer Berlin Heidelberg (2012) 190–206
17. Tisi, M., Perez, S.M., Jouault, F., Cabot, J.: Lazy execution of model-to-model transformations. In: MoDELS. (2011) 32–46
18. Combemale, B., Thirioux, X., Baudry, B.: Formally defining and iterating infinite models. In France, R., Kazmeier, J., Breu, R., Atkinson, C., eds.: Model Driven Engineering Languages and Systems. Volume 7590 of Lecture Notes in Computer Science. Springer Berlin Heidelberg (2012) 119–133
19. Dávid, I., Ráth, I., Varró, D.: Streaming model transformations by complex event processing. In Dingel, J., Schulte, W., Ramos, I., Abrahão, S., Insfran, E., eds.: Model-Driven Engineering Languages and Systems. Volume 8767 of Lecture Notes in Computer Science. Springer International Publishing (2014) 68–83
20. Cuadrado, J.S., Jouault, F., Molina, J.G., Bézivin, J.: Optimization patterns for ocl-based model transformations. In: Models in Software Engineering. Springer (2009) 273–284
21. Beaudoux, O., Blouin, A., Barais, O., Jézéquel, J.M.: Active operations on collections. In: MoDELS. Volume 6394 of LNCS., Springer (2010) 91–105
22. Clavel, M., Egea, M., de Dios, M.A.G.: Building an efficient component for OCL evaluation. ECEASST **15** (2008)
23. Cuadrado, J.S.: A proposal to improve performance of atl collections. MtATL2010 (2010)
24. Jouault, F., Sottet, J.: An AmmA/ATL Solution for the GraBaTs 2009 Reverse Engineering Case Study. In: 5th International Workshop on Graph-Based Tools, Grabats. (2009)

An Adaptable Tool Environment for High-level Differencing of Textual Models

[1]Timo Kehrer, Christopher Pietsch, Udo Kelter
[2]Daniel Strüber, Steffen Vaupel

[1]University of Siegen, Germany
{kehrer,cpietsch,kelter}@informatik.uni-siegen.de
[2]Philipps-Universität Marburg, Germany
{strueber,svaupel}@informatik.uni-marburg.de

Abstract. The use of textual domain-specific modeling languages is an important trend in model-driven software engineering. Just like any other primary development artifact, textual models are subject to continuous change and evolve heavily over time. Consequently, MDE tool chain developers and integrators are faced with the task to select and provide appropriate tools supporting the versioning of textual models. In this paper, we present an adaptable tool environment for high-level differencing of textual models which builds on our previous work on structural model versioning. The approach has been implemented within the SiLift framework and is fully integrated with the Xtext language development framework. We illustrate the adaptability and practicability of the tool environment using a case study which is based on a textual modeling language for simple web applications.

1 Introduction

Model-driven engineering (MDE) is a software development methodology which has gained a lot of interest in many application domains. Besides the MDA initiative and related standards promoted by the OMG, the use of textual domain-specific modeling languages (DMSLs) has emerged as an important trend in modern MDE. Textual DSMLs typically have a small scope and formalize the key concepts of a particular domain of interest. Just like any other primary development artifact, textual models are subject to continuous change and heavily evolve over time. Consequently, appropriate tools supporting standard versioning tasks are strongly required, the calculation of a difference between versions of a model being the most fundamental service.

Selecting a proper versioning tool environment often leads to cost/benefit considerations: On the one hand, one can use off-the-shelf line-based difference tools. This option is attractive since these tools are generic in the sense that they can operate with any kind of textual documents. However, differences are reported on a low level of abstraction and often fail to report complex model changes in a meaningful way. On the other hand, there are sophisticated approaches to structural differencing and merging whose advantages over the classical line-based proceeding are undisputed [25,12]. However, virtually all of these solutions come at a price: many tool components have to be re-implemented for each modeling language anew. Considering the large number of different DSMLs which have to be supported, this often leads to a prohibitive effort.

In this paper, we present a flexible tool environment for high-level differencing of textual models which can be adapted to a new language with moderate effort. The approach builds on our previous work on structural model versioning [18,17,16], which was motivated and has been developed in the context of visual modeling languages. The typical effort to configure a differencing tool ranges between 1 and 10 days, depending on the size of the meta-model. In this paper, we focus on the technical extensions required to support textual models. We argue that a difference tool which is tailored to a given DSML provides significant improvements over existing line-based difference tools. In particular, complex restructurings on a model can be detected, and changes are therefore reported on a higher level of abstraction.

The approach has been implemented within the SiLift framework [27]. It uses several tool components which are based on the Eclipse Modeling Framework (EMF) [8] and is fully integrated with the Xtext language development framework [30]. We illustrate the practicability and adaptability of the tool environment using a case study which is based on a textual modeling language for simple web applications.

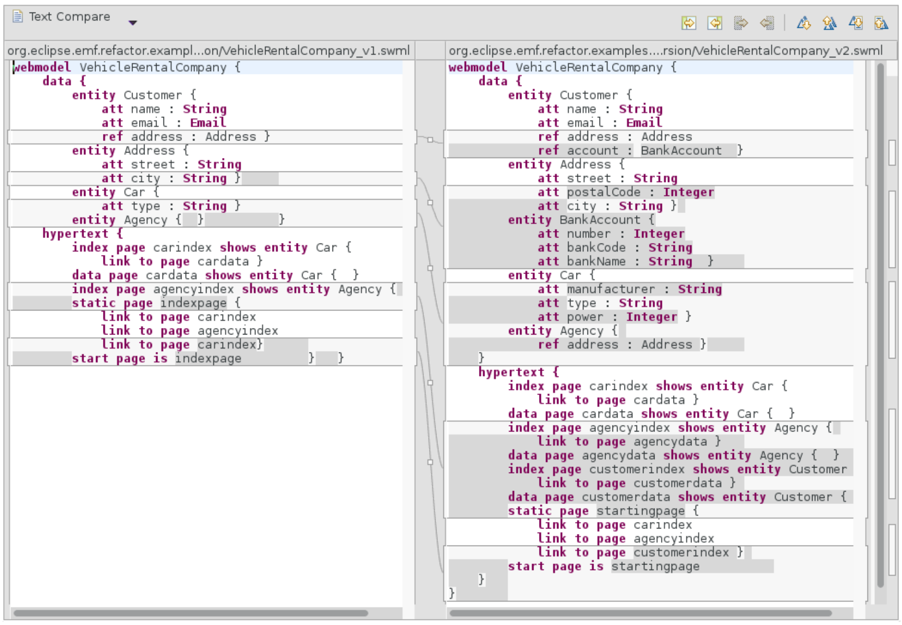
2 Case Study and Motivating Example

In this section, we introduce the case study which will be used to illustrate our approach. The textual modeling language called SWML is introduced in Sec. 2.1. A scenario which describes typical restructurings and improvements on a sample SWML model is described in Sec. 2.2.

2.1 SWML: Simple Web Modeling Language

The Simple Web Modeling Language (SWML) is a textual DSML which aims at defining platform-independent models for a specific kind of web applications. The language has been originally introduced in [7], which also describes a transformation tool chain for generating web applications using standard web development technologies. In this paper, we use the SWML as defined in [5]. In order to keep the paper self-contained, we give an informal description of the SWML abstract syntax:

A WebModel consists of two parts: the DataLayer and the HypertextLayer. The data layer models the application data following basic principles which are known from entity-relationship modeling. An Entity (which is actually an entity type) may have Attributes and References (reference types) to other entity types. Predefined SimpleTypes can be used in attribute declarations. The hypertext layer defines how to present the data using web pages. A Page is either a StaticPage having a fixed content, or a DynamicPage which presents data related to a dedicated entity type. There are two types of dynamic pages: an IndexPage lists the



org.eclipse.emf.refactor.exampl...on/VehicleRentalCompany_v1.swml

```
webmodel VehicleRentalCompany {
    data {
        entity Customer {
            att name : String
            att email : Email
            ref address : Address }
        entity Address {
            att street : String
            att city : String }
        entity Car {
            att type : String }
        entity Agency {  }         }
    hypertext {
        index page carindex shows entity Car {
            link to page cardata }
        data page cardata shows entity Car {  }
        index page agencyindex shows entity Agency {
        static page indexpage {
            link to page carindex
            link to page agencyindex
            link to page carindex}
        start page is indexpage          }   }
```

org.eclipse.emf.refactor.examples....rsion/VehicleRentalCompany_v2.swml

```
webmodel VehicleRentalCompany {
    data {
        entity Customer {
            att name : String
            att email : Email
            ref address : Address
            ref account : BankAccount  }
        entity Address {
            att street : String
            att postalCode : Integer
            att city : String }
        entity BankAccount {
            att number : Integer
            att bankCode : String
            att bankName : String  }
        entity Car {
            att manufacturer : String
            att type : String
            att power : Integer }
        entity Agency {
            ref address : Address }
    }
    hypertext {
        index page carindex shows entity Car {
            link to page cardata }
        data page cardata shows entity Car {  }
        index page agencyindex shows entity Agency {
            link to page agencydata }
        data page agencydata shows entity Agency {  }
        index page customerindex shows entity Customer
            link to page customerdata }
        data page customerdata shows entity Customer {
        static page startingpage {
            link to page carindex
            link to page agencyindex
            link to page customerindex }
        start page is startingpage
    }
}
```

Fig. 1: Initial version v_1 of a sample SWML model and its improved revision v_2

instances of a certain entity type, while a DataPage shows concrete information on a specific entity. The structure of the hypertext layer can be modeled by Links connecting two pages. One of the pages can be declared to be the starting page of the application.

2.2 Example Scenario: Improvements on a Sample SWML Model

SWML models can be conveniently defined using a textual notation. An example model called VehicleRentalCompany taken from [5] is shown in Fig. 1. Version v_1 on the left comes from an early development stage and is used in [5] to illustrate and evaluate quality assurance techniques on textual models. Using metrics and smells as indicators for quality issues concerning the quality aspect completeness, model version v_1 is improved to become version v_2 on the right-hand side of Fig. 1 by applying the following refactorings and manual changes:

1. The smell “No Dynamic Page” for entity type Customer is eliminated by the application of refactoring “Insert Dynamic Pages”: Two dynamic pages (an index page and a data page) referencing entity type Customer are inserted into the hypertext layer. Moreover, the inserted data page is linked by the index page which is in turn linked by the starting page.
2. The smell “Missing Data Page” for index page agencyindex is eliminated by the application of refactoring “Add Data Page to Index Page”: A data page

which shows Agency entities and which is linked by the index page agencyindex is inserted.

3. In order to eliminate the smell "Empty Entity", a new reference address is inserted for entity type Agency.
4. Finally, some missing information is supplemented: A new entity type BankAccount which is referenced by entity type Customer is added to the data layer. Moreover, an attribute postalcode is inserted for entity type Address, attributes manufacturer and power are inserted for entity type Car.

The result of comparing the initial model v_1 and its improved revision v_2 using the Eclipse built-in textual diff utility is shown in Fig. 1. Similar results are obtained using other graphical difference tools such as Meld [24] or KDiff3 [15]. The textual output produced by the UNIX `diff` utility [23] reports 8 deletions and 23 insertions of lines of text. These examples illustrate that the line-based approach fails to explain the improvements on our sample SWML model in an adequate way.

3 High-level Differencing of Textual Models

In this section, we briefly review our approach to high-level model differencing. Next, we describe how to extend the approach and tooling to textual models and finally present our reference implementation which is based on standard Eclipse Modeling technologies and fully integrated with Xtext.

3.1 Approach

In [18], we introduce an approach to high-level differencing which works on a structural representation of two model versions v_1 and v_2 which are to be compared. A model is conceptually regarded as typed, attributed, directed graph which is known as the abstract syntax graph (ASG) of this model. The difference calculation basically proceeds in three steps:

1. Initially, a matching procedure identifies corresponding nodes and edges which are considered to be "the same" elements in v_1 and v_2.
2. Subsequently, a low-level difference is derived. Elements not involved in a correspondence are considered to be deleted or created, each non-identical attribute value of corresponding elements is considered to be updated.
3. Finally, an operation detection algorithm recognizes executions of edit operations in the low-level difference. The available edit operations are provided as additional input parameter, each operation has to be formally specified as a transformation rule in the model transformation language Henshin [4].

Similar to the UNIX `diff` utility, a calculated difference $\Delta(v_1, v_2)$ is a description of how model v_1 can be edited to become revision v_2 in a step-wise manner. However, the available edit operations are defined on the ASG which enables us to report edit steps on a much higher level of abstraction. In principle, any language-specific restructuring operation can be supported as long as it can be specified in a Henshin transformation rule. In other words, we consider the effect of an edit step as an in-place model transformation which is formally specified as a declarative transformation rule to which we refer as edit rule. Thus, the set of available edit operations can be specifically tailored for a given modeling language. An example edit rule for SWML models is briefly explained in Appendix B.

Since the approach presented in [18] has been developed in the context of visual modeling languages, it assumes the allowed types of nodes and edges of an ASG to be defined by a meta-model. Nonetheless, although our approach typically starts with a meta-model, it can be applied to textual DSMLs, too. We only require a procedure which converts the grammar into a meta-model, e.g. as presented in [2,28,6].

3.2 Tool Architecture

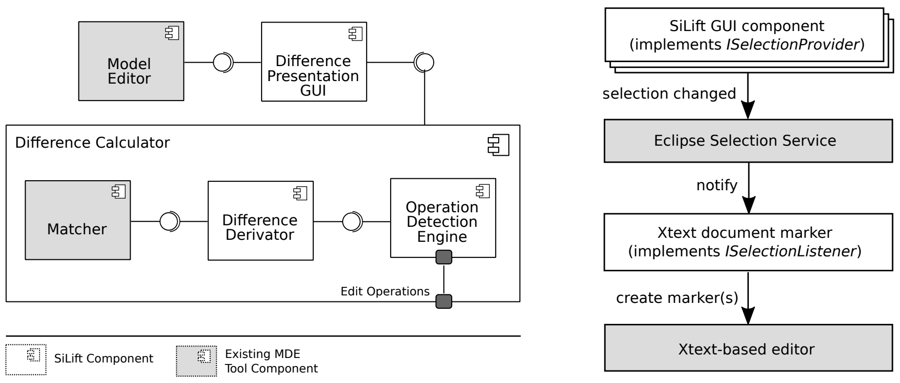
An overview of the core components of a difference tool which implements our approach is shown in Fig. 2a. Exchangeable components which are typically provided by an existing MDE environment are colored in light gray.

The Difference Calculator calculates a difference in a step-wise manner according to our conceptual approach. Consequently, the sub-components Matcher, Difference Derivator and Operation Detection Engine are arranged in a pipeline. A calculated difference is presented to developers in an interactive Difference Presentation GUI as shown in Fig. 3. A control window on the left lists the edit steps. The effect of an edit step is explained on the basis of the concrete syntax. To that end, the original and changed model are displayed in their standard editor on the right. Selecting an edit step in the control window causes the context of this edit step to be highlighted in the respective editor windows. In principle, the GUI can be integrated with any model editor. We only require that the editor offers an API such that external representation of a model element, i.e. certain characters, lines of text or text blocks, can be highlighted.

3.3 Integration with the Xtext Language Development Framework

An EMF-based reference implementation of our approach is available within the model versioning framework SiLift [27]. In this work, we extend the SiLift framework by an integration with the widely used language development environment Xtext [30]. A download option is provided at the accompanying web site of this paper [1].

The adaptation of the algorithmic components is straightforward since an Ecore-based meta-model can be automatically generated by Xtext. The integration of the difference presentation GUI is illustrated in Fig. 2b. The GUI is loosely coupled with generated Xtext editors via the Eclipse Selection Service. All SiLift sub-windows implement the ISelectionProvider interface and thus report which conceptual model elements are currently selected. The selection service notifies registered ISelectionListeners about selection changes induced by a selection


Model Editor
Difference Presentation GUI
Difference Calculator
Matcher
Difference Derivator
Operation Detection Engine
Edit Operations
SiLift Component
Existing MDE Tool Component
SiLift GUI component (implements ISelectionProvider)
selection changed
Eclipse Selection Service
notify
Xtext document marker (implements ISelectionListener)
create marker(s)
Xtext-based editor

(a) Core difference tool components (b) Integration with Xtext

Fig. 2: Tool architecture and integration with the Xtext framework

provider. Our selection listener implementation is based on the Eclipse marker framework which can be used to highlight text fragments in textual Eclipse editors. In order to get the position of a conceptual model element within the textual representation of a model, we utilize the Xtext contribution to the EMF adapter mechanism: For each EObject which originates from an Xtext resource, we get an adapter for this EObject providing access to the corresponding node of the Xtext parse tree. The nodes in a parse tree provide the required position information.

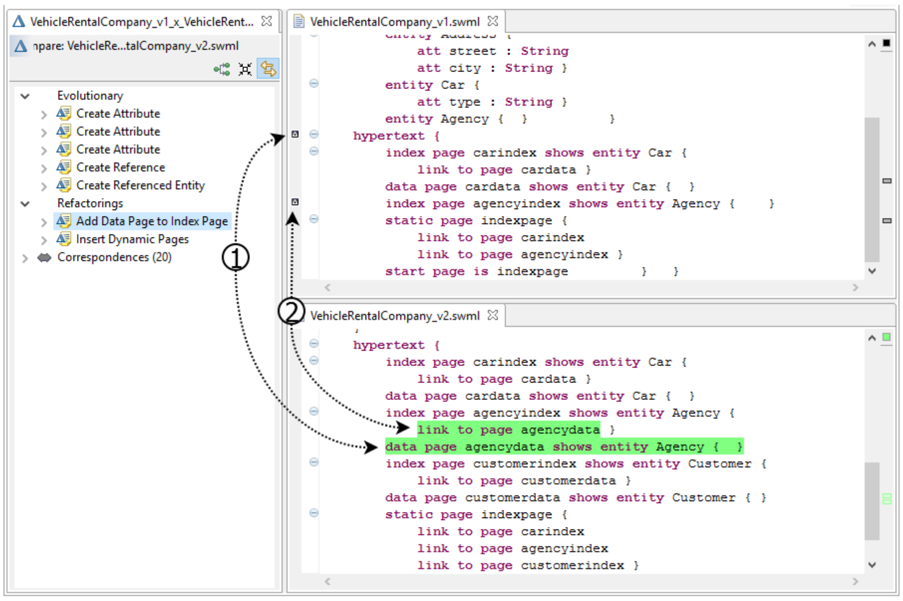
4 Adaptation and Application to SWML

In this section, we outline the adaptation of our tool environment to the SWML. An overview of the difference calculation configuration is given in Sec. 4.1. Finally, Sec. 4.2 presents the results of applying our difference tool which uses this configuration to the example change scenario of Sec. 2.2.

4.1 Configuration of the Difference Calculation

Two of the core differencing components of Fig. 2a have to be adapted to SWML, the matcher and the operation detection engine.

To determine corresponding elements in SWML models, we implemented a signature-based matching strategy [17] using the Epsilon Comparison Language (ECL) [20]. ECL is a domain-specific language for developing highly customized model comparison rules, our SWML matching configuration can be found in the Appendix A. Singleton objects of types WebModel, DataLayer, HypertextLayer are matched immediately. Names of named model elements (Entity, Attribute, etc.) are used as unique signature values, i.e. correspondences are established between


VehicleRentalCompany_v1_x_VehicleRent...
npare: VehicleRe...talCompany_v2.swml
Evolutionary
Create Attribute
Create Attribute
Create Attribute
Create Reference
Create Referenced Entity
Refactorings
Add Data Page to Index Page
Insert Dynamic Pages
Correspondences (20)
1
2
VehicleRentalCompany_v1.swml
VehicleRentalCompany_v2.swml

```
        att street : String
        att city : String }
    entity Car {
        att type : String }
    entity Agency {  }          }
hypertext {
    index page carindex shows entity Car {
        link to page cardata }
    data page cardata shows entity Car {  }
    index page agencyindex shows entity Agency {    }
    static page indexpage {
        link to page carindex
        link to page agencyindex }
    start page is indexpage         }   }
```

```
hypertext {
    index page carindex shows entity Car {
        link to page cardata }
    data page cardata shows entity Car {  }
    index page agencyindex shows entity Agency {
        link to page agencydata }
    data page agencydata shows entity Agency {  }
    index page customerindex shows entity Customer {
        link to page customerdata }
    data page customerdata shows entity Customer { }
    static page indexpage {
        link to page carindex
        link to page agencyindex
        link to page customerindex }
```

Fig. 3: SiLift SWML application: Difference between sample models v_1 and v_2

equally named elements having the same type. Finally, Link objects representing hyperlinks between web pages are matched if they connect the same pages, i.e. if the source and target pages of two links are matched.

The specification of edit rules is supported by one of our meta-tools known as SiDiff Edit Rule Generator (SERGe) [26]. SERGe derives sets of basic edit rules from a given meta-model with multiplicity constraints. These sets are complete in the sense that all kinds of edit rules, i.e. create, delete, move and change operations, are contained for every node type, edge type and attribute defined by the meta-model. For SWML, 29 basic edit rules have been generated. In addition, we manually specified 9 complex edit rules, 5 refactorings which could be re-used from the EMF Refactor tool environment [11], and 4 evolutionary edit operations which facilitate frequently recurring editing tasks. As an example, the edit rule for refactoring "Add Data Page to Index Page", which is applied in step 2 of our motivating example of Sec. 2.2, is shown in Appendix B. The complete set of edit rules is included in the SWML configuration, which is available from the Eclipse update site at [1].

4.2 Application to the Example Change Scenario

Fig. 3 presents the results of applying our difference tool that uses the above configuration to our example of Sec. 2.2. The edit step "Add Data Page to Index Page" is currently selected and its effect can be inspected more closely in the editor windows on the right. The inserted data page agencydata (s. marker indicated by ① in Fig. 3) and the new link to this page from the index page agencyindex (s. marker ② in Fig. 3) are easy to see in the lower editor window which shows version v_2. In the upper editor window showing version v_1, we can see the context of the respective changes, e.g., data page agencydata has been inserted in the hypertext layer. In a similar way, one can interactively inspect the other refactoring "Insert Dynamic Pages" (s. change 1 in Sec. 2.2) and the evolutionary edit steps representing changes 3 and 4 of the change scenario of Sec. 2.2.

5 Related Work

In this section, we briefly review related work regarding the two main aspects of model difference tools addressed in this paper, namely *i)* the adaptability to a new language, and *ii)* the integration with an MDE environment, thereby putting a special emphasis on EMF technologies.

Many approaches and tools to model differencing have been proposed recently, surveys can be found in [13,3]. Similar to ours, virtually all of them work on a structural representation of models. However, only a few of them are adaptable to a new modeling language and almost all of them use primitive graph operations such as creating/deleting single nodes/edges as edit operations for ASGs. The recognition of complex changes such as language-specific refactorings seems to be supported only by few approaches, e.g. [21,22,29]. A detailed review of how these approaches differ from ours can be found in [18,19]. To the best of our knowledge, none of them has yet been adapted to textual DSMLs, which is the main contribution of the tool environment presented in this paper.

A dedicated difference presentation GUI is offered by only a few EMF-based difference tools for models. EMF Compare [9], the currently most widely used differencing tool for EMF-based models, displays two versions of a model in parallel in their abstract syntax tree representation. A similar approach is implemented in EMF Diff/Merge [10] and the RSA tool suite [14]. The parallel display largely fails to present complex model changes. Again, to the best of our knowledge, none of the existing EMF tools can be used with Xtext editors in an integrated way.

6 Conclusion and Future Work

In this paper, we presented concepts and a tool environment to flexibly specify and recognize complex changes in textual models. The tooling, called SiLift, is based on EMF and tightly integrated with the widely used Xtext framework. It enables developers to understand complex structural changes in textual models and is an attractive alternative to traditional line-based difference tools. Moreover, the obtained differences can be converted to executable edit scripts [19] serving as a basis for model patching and structural merging [25].

Obviously, the proposed solutions become more powerful from a practical point of view if they are tightly integrated into an existing version control system such as Git or Subversion. We leave such an integration for future work.

Acknowledgments

This work was partially supported by the DFG (German Research Foundation) under the Priority Programme SPP1593: Design For Future - Managed Software Evolution. Moreover, the authors would like to thank Thorsten Arendt for providing us the quality assurance scenario of the SWML case study.

References

1. Accompanying materials for this paper: http://pi.informatik.uni-siegen.de/projects/SiLift/ocl2015.php; 2015
2. Alanen, M.; Porres, I.: A relation between context-free grammars and meta object facility metamodels; Technical Report No. 606, Turku Centre for Computer Science; 2004
3. Altmanninger, K.; Seidl, M.; Wimmer, M.: A Survey On Model Versioning Approaches; p. 271-304 in: Intl. Journal of Web Information Systems 5:3; 2009
4. Arendt, T.; Biermann, E.; Jurack, S.; Krause, C.; Taentzer, G.: Henshin: Advanced Concepts and Tools for In-Place EMF Model Transformations; in: Proc. MoDELS 2010, LNCS 6394, Springer; 2010
5. Arendt, T.; Taentzer, G.; Weber, A.: Quality Assurance of Textual Models within Eclipse using OCL and Model Transformations; in: Proc. OCL @ MoDELS; 2013
6. Bergmayr, A.; Wimmer, M.: Generating Metamodels from Grammars by Chaining Translational and By-Example Techniques; in: Proc. MDEBE @ MoDELS; 2013
7. Brambilla, M.; Cabot, J.; Wimmer, M.: Model-driven software engineering in practice; p. 1-182 in: Synthesis Lectures on Software Engineering; 1(1); 2012
8. EMF: Eclipse Modeling Framework; http://www.eclipse.org/emf; 2015
9. EMF Compare; http://www.eclipse.org/emf/compare; 2015
10. EMF Diff/Merge; http://eclipse.org/diffmerge; 2015
11. EMF Refactor; https://www.eclipse.org/emf-refactor; 2015
12. Estublier, J.; Leblang, D.; van der Hoek, A.; Conradi, R.; Clemm, G.; Tichy, W.; Wiborg-Weber, D.: Impact of software engineering research on the practice of software configuration management; p. in 383-430: ACM Trans. on Software Engineering and Methodology 14:4; 2005
13. Förtsch, S.; Westfechtel, B.: Differencing and Merging of Software Diagrams - State of the Art and Challenges; p. 90-99 in: Proc. Int. Conf. Software and Data Technologies; 2007
14. IBM Rational Software Architect; http://www.ibm.com/developerworks/rational/products/rsa; 2015
15. KDiff3: http://kdiff3.sourceforge.net; 2014
16. Kehrer, T.; Kelter, U.; Ohrndorf, M.; Sollbach, T.: Understanding Model Evolution through Semantically Lifting Model Differences with SiLift; p. 638-641 in: Proc. 28th IEEE Int. Conf. on Software Maintenance; 2012
17. Kehrer, T.; Kelter, U.; Pietsch, P., Schmidt, M.: Adaptability of Model Comparison Tools; in: Proc. 27th Int. Conf. on Automated Software Engineering; 2012

18. Kehrer, T.; Kelter, U.; Taentzer, G.: A Rule-Based Approach to the Semantic Lifting of Model Differences in the Context of Model Versioning; p. 163-172 in: Proc. 26th Int. Conf. on Automated Software Engineering; 2011
19. Kehrer, T.; Kelter, U.; Taentzer, G.: Consistency-Preserving Edit Scripts in Model Versioning; p.191-201 in: Proc. 28th Int. Conf. on Automated Software Engineering; 2013
20. Kolovos, D.: Establishing Correspondences between Models with the Epsilon Comparison Language; p. 146-157 in: Proc. Intl. Conf. on Model Driven Architecture-Foundations and Applications; 2009
21. Könemann, P.: Capturing the Intention of Model Changes; p.108-122 in: Proc. Int. Conf. on Model Driven Engineering Languages and Systems; 2010
22. Langer, P.; Wimmer, M.; Brosch, P.; Herrmannsdörfer, M.; Seidl, M.; Wieland, K.; Kappel, G.: A posteriori operation detection in evolving software models; p. 551-566 in: Journal of Systems and Software 86(2); 2013
23. MacKenzie, D.; Eggert, P.; Stallman, R.: Comparing and Merging Files with GNU diff and patch; Network Theory Ltd.; 2003
24. Meld: http://meldmerge.org; 2015
25. Mens, T.: A State-of-the-Art Survey on Software Merging; p. 449-462 in: IEEE Trans. Software Eng. 28:5; 2002
26. Rindt, M.; Kehrer, T.; Kelter, U.: Automatic Generation of Consistency-Preserving Edit Operations for MDE Tools; in: Proc. Demonstrations Track of MoDELS; 2014
27. SiLift project; Semantic Lifting of Model Differences; http://pi.informatik.uni-siegen.de/projects/SiLift; 2015
28. Wimmer, M.; Kramler, G.: Bridging grammarware and modelware; in: Satellite Events at the MoDELS 2005 Conference; 2005
29. Xing, Z.; Stroulia, E.: Refactoring detection based on UMLDiff change-facts queries; p.263-274 in: Working Conf. on Reverse Engineering; 2006
30. Xtext; http://eclipse.org/Xtext; 2015

A SWML Matching Configuration Implemented in ECL

```
rule WebModel2WebModel
      match left : Left!WebModel
      with right : Right!WebModel {
      compare {
            return true;
      }
}
// Same for DataLayer and HypertextLayer
// ...

rule Entity2Entity
      match left : Left!Entity
      with right : Right!Entity {
      compare {
            return left.name = right.name;
      }
}
```

```
// Same for Attribute, Reference and Page
// ...

rule Link2Link
	match left : Left!Link
	with right : Right!Link {
	compare {
		return left.srcMatches(right) and left.tgtMatches(right);
	}
}
operation Link srcMatches(other : Link) : Boolean {
	return self.eContainer.name = other.eContainer.name;
}
operation Link tgtMatches(other : Link) : Boolean {
	return self.target.name = other.target.name;
}
```

Listing A-1: SWML Matching Configuration Implemented in ECL

B Refactoring ”Add Data Page to Index Page“ Implemented in Henshin

Fig. 4 shows how to implement the refactoring operation ”Add Data Page to Index Page“ in Henshin. The example illustrates that Henshin offers an intuitive visual syntax to specify model patterns to be found and preserved, to be deleted and to be created. Note that selectedEObject and entityname are input parameters, while New_DataPage and New_Link are output parameters of the rule. The change actions which are to be performed by the rule are specified based on the SWML abstract syntax. Thus, the specification uses type definitions of the SWML meta-model which is generated by the Xtext framework. Given an index page selectedEObject which references an entity named entityname, a new data page New_DataPage referencing this entity is created. Moreover, a new link New_Link is created such that the inserted data page is linked by the index page.

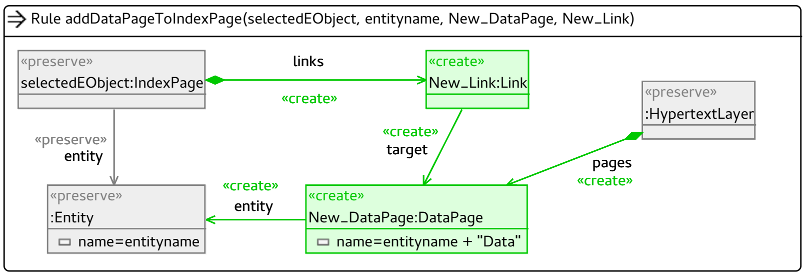


Fig. 4: Refactoring ”Add Data Page to Index Page“ implemented in Henshin

Recursion and Iteration Support in USE Validator with AnATLyzer

Jesús Sánchez Cuadrado

Modelling and Software Engineering Research Group (`http://www.miso.es`)
Universidad Autónoma de Madrid (Spain)

Abstract. Model finders enable numerous verification approaches based on searching the existence of models satisfying certain properties of interest. One of such approaches is anATLyzer, a static analysis tool for ATL transformations, which relies on USE Validator to provide fine grained analysis based on finding witness models that satisfy the OCL path conditions associated to particular errors. However it is limited by the fact that USE Validator does not include built-in support for analysing recursive operations and the *iterate* collection operator.
This paper reports our approach to allow USE Validator to analyse OCL path conditions containing recursive operations and *iterate*, with the aim of widening the amount of actual transformations that can be processed by anATLyzer. We present our approach, based on unfolding recursion into a finite number of steps, and we discuss how to take into account practical aspects such as inheritance and details about the implementation.

Keywords: OCL, ATL, USE Validator, Recursion, Iteration, Model finder, Constraint Solver

1 Introduction

Model finders are an important element of many automated verification approaches in the MDE setting, since they are able to find models satisfying certain properties of interest. Concrete examples of such finders are USE Validator [4] and EMFtoCSP [3], which take as input a meta-model and a set of OCL invariants and return a model satisfying the invariants, if any, within a certain scope (e.g., the maximum number of instantiations of the meta-model classes and ranges of attribute values for infinite types such as integers).

As part of our work in the static analysis of ATL transformations [2], implemented in anATLyzer[1], we have used the model finder implemented by USE Validator [4] to enable the precise analysis of certain error types. This analysis involves creating an OCL path condition which is fed into USE Validator to obtain a model that satisfies it, in order to confirm the error if the model can

[1] http://www.miso.es/tools/anATLyzer.html

be found or to discard the error if not. However, USE Validator does not support recursive operations nor the *iterate* collection operation, hence limiting the applicability of our method in some cases.

This paper reports our approach to enable the analysis of recursive operations and expressions containing the *iterate* collection operation in an OCL-based model finder without support for them, focussing on USE Validator. For recursive operations we unfold the recursion upto a number of levels. In the case of *iterate* we similarly convert each call to a sequence of operations that implements a limited number of iteration steps. We have tested our approach with USE Validator but it could be easily implemented for other systems.
Paper organization. Section 2 introduces the context of this work using an example, Section 3 describes the unfolding of direct recursive operations, whereas Section 4 explains the adaptation of the previous procedure for *iterate*. Finally, Section 5 discusses some issues of our approach and concludes.

2 Context and motivation

The context of this work is our static analysis tool for ATL transformations, called anATLyzer. It consists of a type checking phase in which confirmed failures and potential errors are identified. Then, for each potential error, we compute its OCL path condition, which is an OCL constraint that must be satisfied by any source model that would trigger the error at runtime. Afterwards, such path condition is fed into USE Validator to search for a model, a *witness model*, that satisfies the condition. If found, the error is confirmed, otherwise it is discarded. Hence, a key element for this approach to be practical is to maximise the number of path conditions that can effectively be processed by USE Validator. More details about the approach are described in [2].

As an example let us consider a modified excerpt of the CPL2SPL transformation available in the ATL Zoo[2], which establishes a translation between two telephony DSLs. Figure 1 shows an excerpt of the CPL source meta-model, and an examplary listing[3]. This piece of transformation maps every SubAction source element to a LocalFunctionDeclaration in the target, and each Proxy which satisfies the isSimple predicate into a ReturnStat. In ATL, the relationships between rules are established via *bindings*, denoted by $\leftarrow$, which work as follows. The source elements obtained by evaluating the right part of a binding are looked up in the transformation trace, in order to obtain the corresponding target element created by some rule. In the example, the binding in line 20 is evaluated by executing the expression s.contents.statement which retrieves a Node source element. If such source element has been transformed by some rule, the corresponding target element is assigned to the statement feature.

A smell that the transformation behaviour is not as expected is that a source element appearing in the right part of a binding has not been transformed by

[2] `http://www.eclipse.org/atl/atlTransformations/#CPL2SPL`
[3] We added a filter to the SubAction2Function rule and removed a related rule to make the example more illustrative.

any rule. In this setting, anATLyzer features a rule analysis component which is able to analyse rule–binding relationships to determine if a binding is fully covered by all the resolving rules. To analyse the binding in line 20 anATLyzer builds the OCL path condition shown at the bottom of Figure 1, which states the properties that a model for which the binding would be unresolved must satisfy. This OCL invariant is fed into USE Validator to search for a witness model that confirms the existence of the problem.

However, in practice anATLyzer could not perform this particular analysis due to USE Validator not supporting recursive operations, as is the case of Location.statement. Next section describes how anATLyzer unfolds recursion to enable this analysis, while Section 4 explains how we deal with *iterate*.

3 Direct recursion

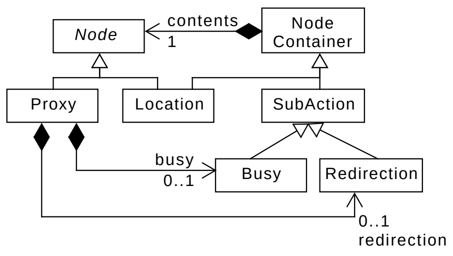
This is the basic recursive schema, in which an operation calls itself in one or more call sites within the operation body. Any OCL specification with an operation featuring even this simple form of recursion is rejected by USE Validator. For the example path condition USE does not try to evaluate the call to Location.statement because it cannot be determined if the operation terminates. Hence, the analysis cannot be carried out.

Our approach to deal with this issue is based on unfolding the recursive operation upto a finite number of steps. We perform the unfolding by copying the original operation n times, so that there are $n + 1$ versions of the operation. Then, each version of the operation is rewritten so that the recursive call sites do not invoke the original operation, but the next copy of the operation. The last operation in the sequence just returns OclUndefined.

Listing 1 shows a sketch of this procedure. It takes the desired number of unfoldings (N) and the piece of abstract syntax corresponding to the recursive operation (OP). There are two helper functions, callSites which returns the set of recursive call sites (i.e., a set of *OperationCall* abstract syntax elements that invoke Op) and copy which returns a deep copy of the given abstract syntax element.

```
N = Number of unfoldings
OP = Original operation

OP$_0$ = OP
for i = 1 to N
  CS$_{i-1}$ = callSites(OP$_{i-1}$)
  foreach cs in CS$_{i-1}$
    cs.operationName = OP.operationName + "_" + i
  end

  OP$_i$ = copy(OP)
  OP$_i$.operationName = OP.operationName + "_" + i
end

OP$_N$.body = OclUndefined
```

Listing 1: Sketch of the unfolding algorithm.



```
 1 -- We consider nodes are statements, by default.
 2 helper context CPL!Node def: statement : CPL!Node =
 3   self;
 4
 5 -- The "location" node is not a statement.
 6 helper context CPL!Location def: statement : CPL!Node =
 7   self.contents.statement;
 8
 9 helper context CPL!Proxy def: isSimple : Boolean =
10   self.busy.oclIsUndefined() and
11   self.redirection.oclIsUndefined();
12
13 rule SubAction2Function {
14   from s : CPL!SubAction (
15             s.contents.oclIsKindOf(CPL!Location) )
16   to t : SPL!LocalFunctionDeclaration (
17       name <- s.id,
18       returnType <- rt,
19       -- Is this binding fully covered by resolving rules?
20       statements <- s.contents.statement
21   )
22 }
23
24 rule Proxy2Return {
25   from s : CPL!Proxy ( s.isSimple )
26   to t : SPL!ReturnStat (
27       ...
28   )
29 }
```

```
SubAction.allInstances()->
  select(s | s.contents.oclIsKindOf(Location))->
  exists(s |
    let _problem_ = s.contents.statement in
      not _problem_.isUndefined() and
      not (if _problem_.oclIsKindOf(Proxy) then
              let s2 = _problem_.oclAsType(Proxy)
               in s2.isSimple()
           else
              false
           endif))
```

Fig. 1: Excerpt of the CPL meta-model (left), two simplified rules of the CPL2SPL transformation (right), and excerpt of the path condition for the problem in line 20 (bottom)

In practice, this procedure needs to be extended to deal with inheritance. This means that it is not enough to duplicate the recursive operation, but every operation that could polymorphically be invoked needs to be duplicated as well.

Listing 2 shows the final result as it is generated to be processed by USE, and complements the OCL path condition presented in Figure 1. Hence, using this method anATLyzer is able to obtain the witness model shown in Figure 2 that confirms the existence of the problem. As can be seen the model contains the elements required to trigger the problem: SubAction and Location objects to trigger the SubAction2Function, and a Proxy element which does not satisfy the isSimple predicate, and it is thus not handled by any rule. Since the Proxy element is linked to SubAction via the contents reference, the binding in line 20 will be unresolved. This model has succesfully been obtained because just two unfolding steps are enough in this case. We heuristically set the maximum number of unfoldings to five, but we still do not have any automated mechanism to set parameter to a safe value for those specific cases in which such reasoning could be possible. For instance, the upper bound of a recursive operation (possibly polymorphic) with no parameters would be the maximum number of instances of the class, and the involved subclasses, set as the the model finder scope.

```
abstract class Node
operations
  statement() : Node = self
  statement_1() : Node = self
  statement_2() : Node = self
  statement_3() : Node = self
end

class Location < Node, NodeContainer
attributes
  url : String
  clear : String
operations
  statement() : Node = self.contents.statement_1()
  statement_1() : Node = self.contents.statement_2()
  statement_2() : Node = self.contents.statement_3()
  statement_3() : Node = OclUndefined
end
```

Listing 2: Unfolded code as generated for USE Validator

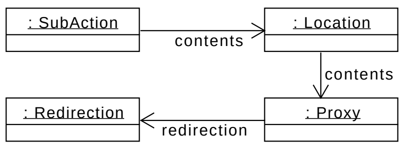


Fig. 2: Witness model obtained for the path condition.

An alternative to this approach is to inline the operation body n times, using *let* expressions to bind parameters. However, we prefer the one presented here because it is easier to handle polymorphic calls as explained.

4 Iterate

The OCL *iterate* collection operation is a general iteration operation with the form col→iterate(it; acc = <init> | <body>). Operationally, it iterates over the elements of the collection assigning them to the it variable in each iteration step.

Each time, the given body is evaluated and the acc variable is updated with the result of the evaluation, so that it has a new value in the following iteration or it is the final result of the operation. As an example, the following code implements the select collection operation in terms of iterate.

```
-- Given: col->select(it | <body>) where col is a Set

col->iterate(it; acc = Set { } |
  if <body> then
    acc->including(it)
  else
    acc
  endif)
```

In practice, USE Validator is able to evaluate most OCL iteration constructs, such as *select*, *any*, etc. However, it cannot evaluate *iterate* which poses a limitation for anATLyzer since path conditions containing *iterate* cannot be processed.

Our approach to deal with this issue is based on unfolding the iteration steps. Until now we support iteration over sets, applying the following strategy. For each call to *iterate* we generate n operations, being n the number of unfoldings, and each of these operations follows the schema shown in Listing 3. First, we check if the collection is empty (line 10) in case the iteration must terminate returning the currently computed value (acc). If the set is not empty, one element is picked using any (line 13), and then a new set is obtained filtering out the picked element from the original set (line 14). To the best of our knowledge this is the only strategy to implement iteration over sets in OCL. Aftewards, the body of the original iterate is evaluated, and the next iteration operation is invoked. Finally, the recursion is ended at depth n by just returning OclUndefined (line 20).

```
-- Given an expression: col->iterate(it : $T_{it}$; acc : $T_{acc}$ = <init> | <body)
-- where:
--       $T_{col}$: the type of the elements of the collection
--       $T_{it}$: the type of the iteration variable, which must be compatible with $T_{col}$
--       $T_{acc}$: the type of the accumulator variable

class ThisModule
operations
  def iterate_aux$_i$(col : Set($T_{col}$), acc : $T_{acc}$) : $T_{acc}$ =
    if col->isEmpty() then
      acc
    else
      let it : $T_{it}$ = col->any(_ | true) in
      let rest : Set($T_{col}$) = col->select(v | v <> it) in
      let value : $T_{col}$ = <body>
       in iterate_aux$_{i+1}$(rest, value)
    endif
  ...

  def iterate_aux$_n$(col : Set($T_{col}$), acc : $T_{acc}$) : $T_{acc}$ = OclUndefined

end
```

Listing 3: Schema for unfolding iterate, using USE syntax

We make use of a special class named ThisModule to allow global operations to be defined. Notably, the iteration operations are defined within this class. In this way, every call to iterate is rewritten to an expression similar to thisModule.iterate_aux0(col, <init>), where the thisModule variable is an instance of ThisModule that must be introduced in the scope of the rewritten expression. We also generate unique identifiers for the iteration operations, to avoid name clashes if there are several calls *iterate* in the same path condition. Finally, we also keep track of the variables in the scope of the original iterate, and if needed, we extend the signature of the iteration operation to pass such variables as parameters.

5 Conclusions and discussion

In this paper we have presented our approach to enable USE Validator analyse recursive operations and the *iterate* collection operation in the context of anATLyzer. In both cases we perform an unfolding of the body of the operations upto a finite number of steps. We have run a small number of tests in which these approaches have shown to be useful, since most of the times a small scope is enough to find the required witness model [1]. Nevertheless, it is part of our future work to carry out more experiments to determine the precision of our approach. In addition, there are some practical considerations to be taken into account, which are discussed in the following.

Given that there is a limit in the number of unfoldings, the last step of the unfolding needs to return some value. Ideally, a *bottom* value should be used to indicate a kind of "stack overflow", in the sense that the recursion has ended prematurely before finishing the computation. In OCL the closest relative to a bottom value is *invalid* which conforms to *OclInvalid*, which in turn conforms to any other type, but any call applied to its unique instance results in invalid itself. However, this is not supported by USE, and thus another value must be used. Selecting such value is difficult in the general case, since it could interact with other expressions processing the return value. We use OclUndefined both for recursion and *iterate* but we are aware that it may affect the accuracy of the solving process.

In this line, an important consequence of unfolding a limited number of times is that the analysis of anATLyzer may not be accurate. A potential error can be wrongly marked as "discarded" only because more unfoldings steps would be needed to provide an accurate answer.

Another issue that affects the accuracy of the approach is that USE only supports sets. Therefore, operations such as *at*, for sequences, cannot be processed. Devising mechanisms to deal with sequences is part of our future work. Hence we aim at studying and adapting other works dealing with these issues, notably the approach proposed in [5] which relies on SMT solving and a more sophisticated unfolding algorithm.

Finally, we have not addressed yet how to unfold mutual recursion, although we believe that a similar strategy is possible, but taking into account the complete call graph of the transformation. This is also part of our future work.

Acknowledgements. This work has been supported by the Spanish MINECO (TIN2011-24139 and TIN2014-52129-R), the R&D programme of the Madrid Region (S2013/ICE-3006), and the EU commission (FP7-ICT-2013-10, #611125).

References

1. A. Andoni, D. Daniliuc, and S. Khurshid. Evaluating the “small scope hypothesis”. Technical Report MIT-LCS-TR-921, MIT CSAIL, 2003.
2. J. S. Cuadrado, E. Guerra, and J. de Lara. Uncovering errors in ATL model transformations using static analysis and constraint solving. In *ISSRE*, pages 34–44. IEEE, 2014.
3. C. A. Gonzalez, F. Büttner, R. Clariso, and 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*, pages 44–50. IEEE Press, 2012.
4. M. Kuhlmann, L. Hamann, and M. Gogolla. Extensive validation of OCL models by integrating SAT solving into USE. In *TOOLS (49)*, volume 6705 of *LNCS*, pages 290–306. Springer, 2011.
5. P. Suter, A. S. Köksal, and V. Kuncak. Satisfiability modulo recursive programs. In *Static Analysis*, pages 298–315. Springer, 2011.

Safe Navigation in OCL

Edward D. Willink[1]

Willink Transformations Ltd, Reading, England,
`ed_at_willink.me.uk`

Abstract. The null object has been useful and troublesome ever since it was introduced. The problems have been mitigated by references in C++, annotations in Java or safe navigation in Groovy, Python and Xbase. Introduction of a safe navigation operator to OCL has some rather unpleasant consequences. We examine these consequences and identify further OCL refinements that are needed to make safe navigation useable.

Keywords: OCL, safe navigation, multiplicity, non-null, null-free

1 Introduction

Tony Hoare apologized in 2009 [3] for inventing the null reference in 1965. This 'billion dollar mistake' has been causing difficulties ever since. However NIL had an earlier existence in LISP and I'm sure many of us would have made the same mistake.

The problem arises because the null object has many, but not all, of the behaviors of an object and any attempt to use one of the missing behaviors leads to a program failure. Perhaps the most obvious missing behavior is used by the navigation expression *anObject.name* which accesses the *name* property of *anObject*. Whenever *anObject* can be null, accessing its *name* property can cause the program to fail.

A reliable program must avoid all navigation failures and so must prove that the source object of every navigation expression is never null. This is often too formidable an undertaking. We are therefore blessed with many programs that fail due to *NullPointerException* when an unanticipated control path is followed.

Language enhancements such as references [2] in C++ allow the non-nullness of objects to be declared as part of the source code. Once these are exploited by good programmers, compile-time analysis can identify a tractably small number of residual navigation hazards that need to be addressed.

A similar capability is available using *@NonNull* [5] annotations in Java, however problems of legacy compatibility for Java's large unannotated libraries makes it very hard to achieve comprehensive detection of null navigation hazards.

An alternative approach is pursued by languages such as Groovy [4], Python [8] and Xbase [10]. A safe navigation operator makes the nulls less dangerous so that *anObject?.name* avoids the failure if *anObject* is null. The failure is replaced by a null result which may solve the problem, or may just move the problem sideways since the program must now be able to handle a null *name*.

In this paper we consider how OCL can combine the static rigor of C++-like references with the dynamic convenience of a safe navigation operator. In Section 2 we introduce the safe navigation operators to OCL and identify that their impact may actually be detrimental. We progressively remedy this in Section 3 by introducing non-null object declarations, null-free collection declarations, null-safe libraries, null-safe models and consider the need for a deep non-null analysis. Finally we briefly consider related work in Section 4 and conclude in Section 5.

2 Safe Navigation Operators

OCL 2.4 [7] has no protection against the navigation of null objects; any such navigation yields an invalid value. This is OCL's way of accommodating a program failure that other languages handle as an exception. OCL provides powerful navigation and collection operators enabling compact expressions such as

```
aPerson.father.name.toUpper()
```

This obviously fails if `aPerson` is null. It also fails whenever `father` is null as may be inevitable in a finite model. A further failure is possible if `name` is null as may happen for an incomplete model.

2.1 Safe Object Navigation Operator

We can easily introduce the safe object navigation operator `?.` to OCL by defining `x?.y` as a short-form for

```
if x <> null then x.y else null endif
```

We can rewrite `aPerson.father.name.toUpper()` for safety as

```
aPerson?.father?.name?.toUpper()
```

This ensures that the result is the expected value or null; no invalid failure.

2.2 Safe Collection Navigation Operator

Collection operations are a very important part of OCL and any collection navigation such as

```
aPerson.children->collect(name)
```

will fail if any element of the `children` collection is null.

We can easily introduce the safe collection navigation operator `?->` to OCL by defining `x?->y` as a short-form for

```
x->excluding(null)->y
```

We can rewrite the problematic collection navigations for safety as:

```
aPerson?.children?->collect(name)
```

This ensures that any null `children` are ignored and so do not cause an invalid failure.

2.3 Safe Implicit-Collect Navigation Operator

The previous example is the long form of explicit collect and so could be written more compactly as:

```
aPerson.children.name
```

The long form of the `?.` operator in `x?.y` is therefore

```
x->excluding(null)->collect(y)
```

We can rewrite for safety as

```
aPerson?.children?.name
```

This again ensures that null `children` are ignored.

2.4 Assessment

OCL 2.4 already has distinct object and collection navigation operators, with implicit-collect and implicit-as-set short-forms. These are sufficient to confuse new or less astute OCL programmers, who may just make a random choice and hope for a tool to correct the choice. Adding a further two operators can only add to the confusion. We must therefore look closely at how tooling can exploit the rigor of OCL to ensure that safe navigation usefully eliminates the null value fragility.

2.5 Safe Navigation Validation

The safe navigation operators should assist in eliminating errors and the following tentative Well Formedness Rules can identify an appropriate choice.

Error: Safe Navigation Required. If the navigation source could be null, a safe navigation operator should be used to avoid a run-time hazard.

Warning: Safe Navigation not Required. If the navigation source cannot be null, a safe navigation operator is unnecessary and may incur run-time overheads.

The critical test is *could-be-null / cannot-be-null*. How do we determine this for OCL?

Some expressions such as constants `42` or `Set{true}` are inherently not null. These can contribute to a program analysis so that a compound expression such as `if ... then Set{42} else Set{} endif` is also non-null even though we may not know anything about the if-condition. Unfortunately, OCL permits any object to be null and so all accesses to objects can be null. In practice this means that most OCL expressions cannot be usefully analyzed and the validation WFRs will just force users to write `?.` everywhere just to silence the irritating errors.

3 Non-null declarations

We have seen how the safe navigation operator is unuseably pessimistic when non-null objects cannot be usefully identified. We will therefore examine how to identify such objects.

3.1 Non-null Object declarations

We could consider introducing non-null declarations analogous to C++ reference declarations. We could even re-use the `&` character. But we don't need to, since UML [9] already provides a solution and a syntax. When declaring a TypedElement, a multiplicity may qualify the type:

```
mandatoryName : String[1]
optionalName : String[?]
```

`[?]` indicates that a String value is optional; a null value is permitted.

`[1]` indicates that a String value is required; a null value is prohibited.

(Other multiplicities such as `[*]` are not appropriate for a single object.).

OCL can exploit this information coming from UML models and may extend the syntax of iterators, let-variables and tuple parts to support similar declarations in OCL expressions. However, since OCL has always permitted nulls, we must treat `[?]` as the default for the extended OCL declarations even though `[1]` is the default for UML declarations.

3.2 Null-free Collection declarations

The ability to declare non-null variables and properties provides some utility for safe navigation validation, but we soon hit another problem. Collection operations are perhaps the most important part of OCL, and any collection may contain none, some or many null elements. Consequently whenever we operate on collection elements we hit the pessimistic could-be-null hazard.

Null objects can often be useful. However collections containing null are rarely useful. The pessimistic could-be-null hazards are therefore doubly annoying for collection elements:

- a large proportion of collection operations are diagnosed as hazardous
- the hazard only exists because the tooling fails to understand typical usage.

In order to eliminate the hazard diagnosis, we must be able to declare that a collection is null-free; i.e. that it contains no null elements. This could be treated as a third boolean qualifier extending the existing ordered and unique qualifiers. We could therefore introduce the new names, NullFreeBag, NullFreeCollection, NullFreeOrderedSet, NullFreeSequence and NullFreeSet but this is beginning to incur combinatorial costs.

A different aspect of UML provides an opportunity for extension. UML supports bounded collections, but OCL does not, even though OCL aspires to UML alignment. The alignment deficiency can be remedied by following a collection declaration by an optional UML multiplicity bound. Thus `Set(String)` is a short-form for `Set(String)[0..*]` allowing UML bounded collections and OCL nested collections to support e.g. `Sequence(Sequence(Integer)[3])[3]` as the declaration of a 3*3 Integer matrix.

However, this UML collection multiplicity tells us nothing about whether elements *cannot-be-null*. We require an extension of the UML collection multiplicity to also declare an element multiplicity. Syntactically we can re-use the vertical bar symbol to allow `[x|y]` to be read as 'a collection of multiplicity x where each element has multiplicity y'. We can now prohibit null elements and null rows by specifying `Sequence(Sequence(Integer)[3|1])[3|1]`.

Finally, we are getting somewhere. A collection operation on a null-free collection obviously has a non-null iterator and so the known non-null elements can propagate throughout complex OCL expressions. Provided we use accurate non-null and null-free declarations in our models, well-written OCL that already avoids null hazards does not need any change. Less well written OCL has its null hazards diagnosed.

3.3 Null-safe libraries

The OCL standard library provides a variety of useful operations and iterations. Their return values may or may not be non-null. The library currently has only semi-formal declarations. These lack the precision we need for null-safe analysis. We will therefore consider how more formal declarations can capture the behaviors that we need to specify.

Simple Declaration Consider the declaration

```
String::toBoolean() : Boolean
```

Using the default legacy interpretation that anything can be null, this should be elaborated as

```
String::toBoolean() : Boolean[?]
```

But we have an additional postcondition:

```
post: result = (self = 'true')
```

Intuitively this assures a true/false result. But we must always consider null and invalid carefully. If `self` is null, the comparison using `OclAny::=` returns false, and if `self` is invalid the result is invalid. We are therefore able to provide a stronger backward compatible library declaration that guarantees a non-null result.

```
String::toBoolean() : Boolean[1]
```

We can pursue similar reasoning to provide `[?]` and `[1]` throughout the standard library.

Complex Declaration We hit problems where the non-null-ness/null-free-ness of a result is dependent on the non-null-ness/null-free-ness of one or more inputs.

Consider a declaration for `Set::including` in which we use parameters such as `T1`, `c1`, `e1` to represent flexibilities that we may need to constrain.

```
Set(T1)[c1|e1]::including(T2)(x2 : T2[e2]) : Set(T3)[c3|e3]
```

The relationship between `T1`, `T2` and `T3` is not clear in the current OCL specification. Some implementations emulate Java-style collection declarations where the result is the modified input; `T3` is therefore the same as `T1`, and `T2` must be assignable to `T1`. This implementation-driven restriction is not necessary for a declarative specification language such as OCL where we just require that each of `T1` and `T2` are assignable to `T3`. The declarative flexibility can be captured by a single type parameter and a direction that the most derived solution be selected from the many possible solutions.

```
Set(T)[c1|e1]::including(x2 : T[e2]) : Set(T)[c3|e3]
```

The result is only null-free if the input collection is null-free and the additional value is non-null. Therefore if `e1` and `e2` are Boolean-valued with true for `[1]` (is not null) and false for `[?]` (may be null), `e3` may be computed as:

```
e3 = e1 and e2
```

This computation can be included in a library model to avoid the need for an implementation to transliterate specification words into code.

We can also compute `c3` pessimistically as

```
    c3.lower = c1.lower
    c3.upper = if c1.upper = * then * else c1.upper+1 endif
```

Preliminary discussions at Aachen [1] indicated limited enthusiasm for accurate modeling of collection bounds in OCL, so we could just take the view that OCL does not support bounded collections enthusiastically; The definition of `c3` is then much simpler:

```
    c3.lower = 0
    c3.upper = *
```

However if we need accurate equations to avoid loss of non-null-ness precision for library operations, the simplification of not providing similar equations for collection bounds may prove to be a false saving.

3.4 Null-safe models

Once the standard library has accurate null-safe modeling we are just left with the problem of user models.

For object declarations, there seems little choice but to make this part of the user's modeling discipline; object declarations must accurately permit or prohibit the use of null.

For collection declarations the default may-be-null legacy behavior is mostly wrong and for some users it may be universally wrong. We would like to provide a universal change to the default so that all collections are null-free unless explicitly declared to be null-full. In UML, we can achieve this by defining an `OCL::Collections::nullFree` stereotype property for a Package or Class. The `nullFree` Boolean property provides a setting that is 'inherited' by all collection-valued properties within the Package or Class.

UML has no support for declaring collection elements to be non-null, so we need a further `OCL::Collection::nullFree` stereotype property to define whether an individual TypedElement has a null-free collection or not.

For disciplined modelers, the sole cost of migrating to null-safe OCL will be to apply an OCL::Collections stereotype to each of their Packages.

Feedback from workshop UML is moving, and perhaps has already moved, to prohibit nulls in multi-valued properties. UML-derived collections are therefore inherently null-free and no stereotype is required. Rather the converse of a null-full declaration is needed to declare that nulls are really required and that some workaround for the UML prohibition is to be used.

3.5 Deep non-null analysis

Accurate non-null declarations enable WFRs to diagnose null navigation hazards ensuring that safe navigation is used when necessary. However simple WFRs provide pessimistic analysis.

For instance, the `anObject.name` navigation in the following example is safe since it is guarded by `anObject <> null`

```
let anObject : NamedElement[?] = ....
in anObject <> null implies anObject.name <> null
```

However a simple WFR using just `anObject : NamedElement[?]` diagnoses a lack of safety because the `anObject` let-variable may be null. A potentially exponential program flow analysis is needed to eliminate all possible false unsafe diagnostics. A simpler pragmatic program flow analysis can eliminate the common cases of an if/implies/and non-null guard.

4 Related Work

The origin and long history of null problems has been alluded to in the introduction as has the mitigation for C++ and Java.

The safe navigation operator is not new since at least Groovy, Python and Xbase provide it.

The database usage of NULL as an absence of value is in principle similar to OCL's use of null, however whereas use of null in OCL leads to failures, SQL is more forgiving. This can be helpful, but also hazardous.

The possibility of safe navigation in OCL is new, or rather the pair of `?.` and `?->` operators were new when we suggested them at the Aachen workshop [1]. The utility of the `[?]` and `[1]` non-null multiplicities was also mentioned at the Aachen workshop. The null-free declarations, stereotypes and the interaction between safe navigation and non-null multiplicities have not been presented before, although they are available in the Mars release of Eclipse OCL [6].

5 Conclusions

We find that naive introduction of safe navigation to OCL risks just doubling the number of arbitrary navigation operator choices for an unskilled OCL user. These problems are soluble with tool support provided we can also solve the problem of declaring non-null objects and null-free collections.

We take inspiration from UML multiplicity declarations to provide the necessary declarations. We use stereotypes for declarations that are not inherently supported by UML.

The cost for well-designed models may be as little as

– one stereotype per Package to specify that all of its collections are null-free
– an accurate `[?]` or `[1]` multiplicity to encode the design intent of each non-collection Property

The benefit is that OCL navigation can be fully checked for null safety.

Acknowledgments Many thanks to Adolfo Sánchez-Barbudo Herrera for his detailed review and constructive comments.

References

1. Brucker, A., Chiorean, D., Clark, T., Demuth, B., Gogolla, M., Plotnikov, D., Rumpe, B., Willink, E., Wolff, B.: Report on the Aachen OCL Meeting . CEUR-WS Proceedings, Vol-1092 (2013) http://ceur-ws.org/Vol-1092/aachen.pdf
2. Ellis,M., Stroutstrup, B.: The Annotated C++ Reference Manual. (1990)
3. Hoare, T.: Null References: The Billion Dollar Mistake. QCon London (2009)
4. JSR 241: The Groovy Programming Language. (2004)
5. Using null annotations: http://help.eclipse.org/luna/index.jsp?topic=%2Forg.eclipse.jdt.doc.user%2Ftasks%2Ftask-using%5Fnull%5Fannotations.htm
6. Eclipse OCL: https://www.eclipse.org/modeling/mdt/downloads/?project=ocl
7. Object Constraint Language. Version 2.4., OMG Document Number: formal/2014-02-03, Object Management Group (2009), http://www.omg.org/spec/OCL/2.4
8. Python Software Foundation: The Python Language Reference. 2.7.10 (2015)
9. OMG Unified Modeling Language (OMG UML), Version 2.5, OMG Document Number: formal/15-03-01, Object Management Group (2015), http://www.omg.org/spec/UML/2.5
10. Xbase: https://www.eclipse.org/Xtext/documentation/305%5Fxbase.html#xbase-expressions

Tool Paper: A Lightweight Formal Encoding of a Constraint Language for DSMLs

Arnaud Dieumegard, Marc Pantel, Guillaume Babin, and Martin Carton

IRIT ENSEEIHT
Université de Toulouse
France
`first.last@enseeiht.fr`

Abstract. Domain Specific Modeling Languages (DSMLs) plays a key role in the development of Safety Critical Systems to model system requirements and implementation. They often need to integrate property and query sub-languages. As a standardized modeling language, OCL can play a key role in their definition as they can rely both on its concepts and textual syntax which are well known in the Model Driven Engineering community. For example, most DSMLs are defined using MOF for their abstract syntax and OCL for their static semantics as a metamodeling DSML. OCLinEcore in the Eclipse platform is an example of such a metamodeling DSML integrating OCL as a language component in order to benefit from its property and query facilities. DSMLs for Safety Critical Systems usually provide formal model verification activities for checking models completeness or consistency, and implementation correctness with respect to requirements. This contribution describes a framework to ease the definition of such formal verification tools by relying on a common translation from a subset of OCL to the WHY3 verification toolset. This subset was selected to ease efficient automated verification. This framework is illustrated using a block specification language for data flow languages where a subset of OCL is used as a component language.

1 Introduction

Domain Specific Modeling Languages (DSMLs) are used in many domains where their capabilities were shown to be very useful for assistance in system and software engineering activities like requirements analysis, design, automated generation, validation or verification. The critical system and software industry is one of the domains where they have been used for many years. This has allowed to achieve better quality and safety of the products without having their development cost following exponentially the curve of the systems complexity.

Beside the obvious advantages of relying on DSML's for the formalization of requirements and design elements, and the automation or simplification of these activities, it remains difficult to convince at the certification level on their correctness. Formal methods have been used in this purpose and as such have been proven as being formidable allies, especially for the verification and validation tasks. In the recent upgrade to level C of aeronautics standards DO-178, both technologies were introduced as DO-331 for Model Driven Engineering and DO-333 for Formal Methods. Both documents advocate DSMLs use but their combined use given in DO-333 looks more promising that the lone use of DSMLs in DO-331. Both documents advocate a precise, complete and non-ambiguous specification of DSMLs.

Their specification usually relies first on the model based definition of their concepts using a DSML such as the OMG Meta Object Facility (MOF). These metamodels are usually completed with static semantics expressed with constraint languages such as the OMG Object Constraint Language (OCL) that provides first order logic and model navigation construct. It allows specifying the static properties of the DSML itself and its concepts. Many formal specifications of both MOF and OCL have been conducted to assess their properties, validate the standards and participate in the verification of their implementation. These specifications may also provide formal verification tools for models expressed with these languages. However, this was usually not their main purpose and thus these tools do not usually behave well on the scalability side. This contribution targets a more scalable verification dedicated encoding of a subset of OCL in the WHY3 verification toolset that can be reused in different DSMLs. These DSMLs rely on this subset of OCL as a component language. One key change is that the data part of OCL must be adapted to fit the host language.

The WHY and WHYML languages are two complementary languages used for formal software verification. The first one is a high level specification language and the second one a powerful programming language (WHYML). In this setting, programs written using the latter are relying on the formal definitions for types, theories and functions prototypes defined with the former. The WHY3 toolset provides a harness for the manipulation of both specifications and programs and also for their translation in order to be verified either automatically using SMT solvers or manually using proof assistants like COQ, PVS or ISABELLE. The WHY3 toolset is very powerful in the sense that it allows to benefit from the combined strengths of each managed SMT solver and if necessary to rely on manual verification for proofs that may be too difficult to achieve automatically.

By providing a formalisation of a subset of the OCL in the WHY3 toolset, we target to build a framework allowing to ease the formal verification of DSMLs implementations and instances. In order to do so, we propose a formal specification and implementation of a lightweight version of the OCL language. DSMLs specification, expressed using for example MOF compliant metamodels and OCL constraints, or their instances can then be translated to the WHY language and be automatically and formally verified. Such verification can ensure the correctness of the DSML specification, assess the conformance of its instances, or verify complex properties on the instances. We have applied this approach for the verification of correctness properties over the instances of a data flow languages specification DSML [10].

This contribution first pictures our use case and its verification in Section 2. We then provide in Section 3 our lightweight formal encoding of the selected subset of OCL using the WHY and WHYML languages. We refer to it as lightweight as it does not covers the whole OCL in order to have a better verification automation and scalability. We will additionally discuss the coverage of the OCL specification. We compare our approach to already existing ones in Section 4. We finally conclude and detail some of our perspectives in Section 5.

2 OCL as a DSML Component

As an Object Constraint Language, OCL has been mainly used to define properties to be verified on class instances in UML. It thus includes first order logic and query facilities over class instances and its data part is strongly related to the UML one. In the OMG MDA proposal, DSMLs were first defined as simplified class diagrams in the MOF subset of UML. OCL has then been used to model the static semantics of the DSML elements. Nowadays, most OMG standards rely both on MOF and OCL as specification languages.

These ones are thus now widely known in the software engineering domain. They can then be used as a language component to be included directly in the DSML [11] to allow the DSML user to model constraints and queries with a well known and standard notation. OMG already enforces the reuse of its languages like OCL in the QVT standard. Other DSMLs have integrated OCL as a language component like the use of TOCL for the specification of common behavioral requirements for EMF DSML models in[21] or for execution trace matching in the FEVEREL DSML [20], the Event Constraint Language (ECL) which allows to derive CCSL constraints from EMF DSML models [8], the Behavioral Coordination Operator Language which allows to specify executable model coordination operators[14]. These DSMLs metamodels combine several metamodel components including the OCL one that relies on identifier to connect to the elements in the other metamodel components. In the usual integration of OCL in UML or MOF, the OCL expressions are written in contexts that provides the identifiers from the UML/MOF part of the DSML and the data part of OCL is mapped to their data parts. The identifiers can then be used in the OCL expressions like the names of classes, attributes, methods and parameters. OCL itself can define identifiers that bind stronger than the ones from the context. A similar approach is used to connect OCL with other parts in the DSML metamodel. One key point is to connect also the OCL data part with the DSML one.

2.1 General approach for the verification of DSMLs

Formalisation of a DSML in order to conduct formal verification activities starts with the formalisation of the DSML structural elements and their properties. It is usually achieved by expressing the DSML structure, and its static semantics (the metamodel and its OCL constraints) with the formalism targeted for the verification. Two approaches are commonly used in that purpose: on the one hand, modeling the DSML in the formal verification platform including the model creation facilities and then building and verifying models directly in such a platform (a kind of denotational semantics); and on the other hand, providing a transformation from the DSML to the verification platform that is applied on each model for verification (a kind of translational semantics). The first one builds the formal model at the language level, whereas the second one works at the instance level. In both cases, the writing (either being manual or automated) of the formal representation is parametrized by the DSML data specification (i.e. the data that are expressed in the models and verified by the constraints).

If the DSML relies on an additional language such as OCL as a component to express properties, it is then also required to translate those properties into the verification formalism. This work can be leveraged by first providing a formalisation for the OCL component language and then relying on this formalisation for the automatic translation of the other parts of the DSML.

Many DSML also include a definition for their execution semantics. This definition can be directly embedded in the DSML definition[5]; it can also be defined externally or can be provided using third party component action languages such as ALF[1]. The execution semantics should also be expressed using the verification formalism and the associated formal verification platform. This provides an execution semantics formal specification. This one can be manually or automatically translated into the formal verification platform.

Finally, the verification of a DSML is based on the expression of properties to be verified. These properties are either written directly using the verification formalism or automatically generated from the DSML instances. The second approach is preferred as it increase the efficiency of the verification and also eases the verification for the DSML user.

In the following, we illustrate the verification of a DSML through the example of the verification of the BlockLibrary DSML. This DSML allows using a limited subset of OCL as a component language. BlockLibrary DSML instances are translated as Why3 theories. OCL expressions are translated to Why3 expressions relying on the pre-existing formalisation of the restricted version of the language. Finally, high level properties are automatically generated from the DSML instances as Why3 goals to be discharged through the Why toolset.

2.2 The BlockLibrary DSML structural specification

The BlockLibrary DSML [9, 10] aims at the specification of the structure and semantics of data flow atomic blocks such as the ones at the core of the Simulink or Scade languages. This DSML allows to specify the inherent variability of these blocks structure and semantics. The structural specification part of the DSML is inspired from the concepts of software product lines combined with the object oriented concept of inheritance in order to handle the highly variable nature of the blocks specification. Data types can be attached to blocks specification components (inputs, outputs, parameters and memories). The components are specified with constraints describing their allowed values (specified using a subset of OCL). The semantics of the blocks is expressed with a simple action language that relies on the same subset of OCL to express pre/post-conditions, variants and invariants.

[1] http://www.omg.org/spec/ALF/Current

The two main concerns in the DSML are the use of OCL as a language component for the specification of the DSML data and action constraints and the variability management that allows combining parts of the blocks specification into multiple block configurations.

2.3 Verification of BlockLibrary instances with Why3

A significant part of the effort in this case study has been focused on the formalisation of the DSML constructs as WHY theories to conduct formal proofs of the model completeness, consistency and correctness using the WHY3 platform.

A block specification using the BlockLibrary dsml We provide in Figure 1 a simplified specification for the `Delay` block written using the BLOCKLIBRARY DSML. The `Delay` block outputs the value provided as its first input and delays its value by a certain number of clock ticks. The number of clock ticks is either provided as a parameter of the block or as an additional input. The values of the output of the block for the first clock ticks cannot be computed from the inputs of the block and should thus be provided as either a parameter or as an additional input of the block. The block allows for scalar, vectors and matrix values. The specification provided here is limited to the handling of scalar values and the initial condition can only be provided as a parameter of the block.

A block specification is contained in a `blocktype` element bundled in a `Library` construct. The data types used in the specification are declared in the `Library`. The `variant` constructs contains the specification for a block structural elements: input (`in`) and output (`out`) ports, `parameters` and `memories`. For each declaration of a structural element, it is possible to specify its default value and some additional constraints using our subset of the OCL language. We provide such a constraint in line 20, where we specify that the `Delay parameter` can only have a positive value.

Each `variant` can extend, in an object oriented way, other `variant` constructs by relying on the `extends` construct. This composition is done through two n-ary operators that can be applied on `variant` constructs: `allof` and `oneof`. The former specifies mandatory compositions whereas the former specify alternative compositions. These operators are inspired from the software product line approach for the specification of features hierarchies.

The `mode` constructs allows for the specification of the blocks semantics variation points. They thus provide an implementation for one or more `variant` constructs. The previously presented composition language can be used in this purpose. Each `mode` construct must provide a definition for the `compute` semantics of the block variation point (lines 57 to 61). It may also provide the optional semantics for the `init` and `update` computation phases for the specified block. The blocks semantics are provided using a custom simple action language: the BLOCKLIBRARY action language (`bal`). A `mode` and its implemented `variant` make a set of block configurations. This set is extracted from the composition of `variant` elements through the `allof` and `oneof` constructs.

```
library Lib {

type signed realInt TInt32 of 32 bits
type realDouble TDouble
type array TArrayDouble of TDouble

blocktype DelayBlock {
  variant InputScalar {
    in data Input : TDouble
    out data Output : TDouble
  }
  variant ICScalar {
    parameter IC : TDouble default 0.0
  }
  variant ICVector {
    parameter IC : TArrayDouble
  }
  variant InternalDelay {
    parameter Delay : TInt32 {
      invariant ocl { Delay.value > 0 }
    }
  }
  variant ExternalDelay {
    in data Delay : TInt32 {
      invariant ocl { Delay.value > 0 }
    }
  }
  variant ListDelay_Scalar extends allof (
    oneof (InternalDelay, ExternalDelay),
    InputScalar, ICVector
  ) {
    invariant ocl { Delay.value > 1 }
    invariant ocl {
      IC.value.size() = Delay.value
    }
    memory Mem {
      datatype auto ocl {Input.value}
      length auto ocl {DelayUpperBound.value}
    }
  }
  variant Delay_Scalar extends allof (
    oneof (InternalDelay, ExternalDelay),
    InputScalar, ICScalar
  ) {
    invariant ocl { Delay.value = 1 }
    memory Mem {
      datatype auto ocl {Input.value}
      length auto ocl {1}
    }
  }
  mode DelayMode_Simple implements Delay_Scalar
      {
    init init_Delay_Simple bal {
      postcondition ocl {
        Mem.value = IC.value }
      Mem.value = IC.value;
    }
    compute compute_Delay_Simple bal {
      postcondition ocl {
        Output.value = Mem.value }
      Output.value = Mem.value;
    }
    update update_Delay_Simple bal {
      postcondition ocl {
        Mem.value = Input.value }
      Mem.value = Input.value;
    }
  }
  mode DelayMode_List implements
      ListDelay_Scalar{
    init init_Delay_List bal {
      postcondition ocl {
        Mem.value = IC.value }
      Mem.value = IC.value;
    }
    compute compute_Delay_List bal {
      postcondition ocl {
        Output.value = Mem.value->first() }
      Output.value = Mem.value[0];
    }
    update update_Delay_List bal {
      postcondition ocl { Mem.value =
        Mem.value
        ->excluding(Mem.value->first())
        ->append(Input.value)
      }
      for (var i=0; i < Mem.value->size()-1;
        i = i+1){
        Mem.value[i] = Mem.value[i+1];
      }
      Mem.value[Delay.value-1] = Input.value;
    }
  }
}}
```

Fig. 1. Delay block specification using the BlockLibrary dsml

The lack of space prevents us to detail further this dsml and we refer the reader to our papers[9, 10], the related thesis work [2], and the website[3] for additional informations.

BlockLibrary specification verification A transformation translates BlockLibrary instances to Why theories; ocl-like constraints to predicates relying

[2] http://www.dieumegard.net/thesis/Thesis-Arnaud_Dieumegard.pdf
[3] http://block-library.enseeiht.fr

on a lightweight formalisation of the subset of OCL detailed in Section 3; and semantics specification (specified using the action language) to WhyML functions.

In addition to this translation of the BlockLibrary specification, properties are generated to assess: the completeness (are all possible configurations of a block given in the specification ?) and the disjointness (are all possible configurations of a block expressed only once in the specification ?) of all the BlockLibrary DSML block specifications. The verification of these two properties is done automatically by SMT solvers in a few tenth of a second for rather simple blocks specification to up to a few seconds for mode complex ones containing up to a thousand different configurations for a block.

The most important lesson learned from this experiment was that as soon as the DSML constructs, their attached constraints, and the OCL component expressions are translated to Why, it is straightforward to write or generate high level properties to be verified using the Why and WhyML languages.

The following section summarizes the Why model of the specific subset of the OCL that was necessary for our DSML use case where a subset of the OCL is used as a DSML component.

3 OCL Formalisation Using the Why Toolset

OCL is a formal language for querying model elements and expressing model properties. It relies on first order logic and model traversal operators. It is guaranteed to be side-effect free and as such cannot modify the model it is applied to. The Why language is more expressive than OCL and can thus be used to encode any OCL construct.

We provide here some details on our formal specification of our specific subset of OCL in Why. It is important to keep in mind that the main objective of the BlockLibrary DSML is to allow for the formal specification of blocks. As such, we enforce the block specifier to provide strictly typed elements. The consequences of this limitation is impacting the subset of the OCL to be handled. It is indeed not required to handle type manipulation operations such as *oclIsKindOf* or *oclAsType* among others.

We will go through the support for some of the OCL constructs; the limitations we selected; and the translation strategies in order to go from OCL constructs to Why constructs. As we will not provide the complete details about our translation, we invite the interested reader to refer to our website[4] for additional informations.

3.1 OCL standard data types and collections

OCL is built on a simple set of types called primitive data types. The Why3 standard library provides a formalisation for primitive data types that largely covers the needs for the mapping of the OCL ones.

[4] http://block-library.enseeiht.fr/html

OCL primitive values can be gathered in collections that differs regarding their ability to handle multiple occurrences of the same value (*Bag*, *Sequence*) or not (*Set*, *OrderedSet*) and if these values are ordered (*Sequence*, *OrderedSet*) or not (*Bag*, *Set*).

In the WHY standard library, collections can be modeled as lists allowing multiple occurrences of the same unordered value. This makes them a direct translation for *Bag* collections. The standard library also provides the support for Arrays that would directly map to the *Sequence* collection and Sets for *Set* collection in OCL.

In our implementation of OCL, we do not take into account the type of the collections and only provide support for *Bag* collections as lists. This allows to simplify their management in our implementation as it removes the side effects on the collection management operations such as that elements are not automatically removed from the collections as multiple occurrences are allowed. We defined a WHY theory called *OCLCollectionOperation* containing the definition for some basic list accessors and operations. The other three kinds of OCL collections will be implemented in a similar way in a near future with no specific issues expected.

We decided not to support messaging related constructs and tuples constructs. Messaging is related to UML sequence and state machine diagrams which was out of the scope of our case study. Tuples are also missing in our implementation and could be implemented on a first approximation by relying on record types in WHY.

3.2 OCL operations translation strategy

OCL defines multiple operations that are to be applied either on primitive values – referred to as standard language operations – or on collection values – referred to as collection and iteration operations. Some of these operations are not supposed to be used on *Bag* collections and explicit conversions between collections types must be done. We do not enforce this currently in our implementation of OCL as we provide only one type of collection.

The translation of OCL expressions to the WHY language can be done using two strategies. First, operations can be translated to basic first order logic expressions and thus can directly be used in WHY. Second, the definition for the operations can be axiomatized using WHY function declarations and OCL constraints are translated as expressions using these functions. In our work, we rely on a combination of the two. The list getter is used for simple collection accesses, standard type operations have been defined as functions in WHY, simple collections operations are directly mapped to their logical expression equivalent.

This approach has the advantages of easing the translation work as the semantics of OCL standard types operations is already defined and thus avoid to generate too complex expressions. This has the pleasant side effect of easing the transformation verification activities as the translation itself is simpler.

In the following, we provide the mapping between the source OCL constructs and operations and the target WHY predicates, functions and expressions.

3.3 OCL standard library operations

In OCL expressions, operations can be applied on primitive OCL types. These operations are classical handling of primitive data types and are gathered in the OCL standard library. They have already been formalized in the Why library. We thus rely on their formalisation for our translation.

We did not currently implement the transformation for the *div* and *mod* operations on OCL *Double* elements. The *div* operation is of particular interest as its behavior in OCL and in Why are not the same. Indeed where the OCL implementation of *div* applied to any number and 0 (division by zero) returns the *null* value, the Why version is simply not defined and rejected by the formal verification toolset. The management of specific values like *null* and *invalid* is not done and is a chosen limitation of our work to simplify the formal models and reduce the verification costs by avoiding three value logics (*True*, *False* and *invalid*). We give later more details regarding the related restrictions.

Boolean OCL expressions are implemented using boolean expressions in Why. *xor* operator has been implemented with *and*, *or* and *not* operators. Numeric operations have been implemented using the standard Why arithmetic theories and their operators. Finally relational operators are also based on standard Why constructs.

3.4 Collection operations

As previously mentioned, there are four types of OCL collections. We only considered the use of the *bag* collections in our case study. In our handling of OCL collection operations, we do not handle OCL generic nature nor subtyping of elements. If the same operation is to be expressed on different types, it is then developed separately for each different type. Whereas this may seem to be a limited way of handling this problem, in practice, our supported restriction of OCL makes this easier as only a few operations needs to be encoded several times. Both kind of polymorphism could be handled by synthesizing sum types representing the allowed set of types for an operation and appropriate pattern matching that mimics late binding at a higher verification cost.

Our partial handling of OCL collections has an impact on the translation provided for some collection operations. The *append* and *including* operations have the same implementations and so are *subOrderedSet* and *subSequence.* According to the OCL specification, some collections operations are not allowed on *bag* collections: *append*, *at*, *first*, *indexOf*, *indexAt*, *last*, *prepend*, *subOrderedSet*, *subSequence.* We decided to allow their use in our implementation of OCL. This is a significant drift from the OCL standard but it has the advantage of greatly simplifying the translation mechanism without restraining the expressiveness of the language. This restriction could be enforced easily at the static semantics level. The formalisation of the usual OCL collections is of additional complexity as studied by Mentré et Al [16] but was not of primary interest for our current work so we decided not to address the related issues.

Each of these functions are defined through a set of axioms specifying their context of use: on which element type they are defined; what restrictions are required for their definitions; and the result of their computation according to the provided input values. The restrictions on their parameters are used to avoid the *invalid* value. The effort needed in order to achieve the complete verification remains important but is highly lightened by the use of SMT solvers to automate the verification.

3.5 Logical property assessment iteration operations

Iteration operations are the main operations used on OCL collections. They allow to assess a logical property verification on: every element of a list (forAll), at least one element of a list (exists), exactly one element of a list (one). They can also express the uniqueness of the result of the application of a function on every element of a list (isUnique). All these operations returns a boolean value.

In Table 1, we provide the translation for the *forAll*, *exists*, *one* and *isUnique* OCL operations on collections. Unlike the previous translations, we do not rely on predefined functions but we rather map these operations to simple first order logic expressions. Regarding the translation of the condition expression: *exp*, the defined OCL iterators: *it*, *it1* and *it2* are mapped to a call to the position of the element in the collection via the list getter operator. In practice, references to *it* or *it1* are replaced by *a[i]* and references to *it2* are replaced by *a[j]* in *exp*. This is expressed using the function application: $[a/b]c$ that substitute a to b in c.

ocl expression	Target Why code
a→**forAll**(it: DT | exp)	∀ i: int. 0 ≤ i < length a → [a[i]/it]exp
a→**forAll**(it1, it2: DT | exp)	∀ i j: int. 0 ≤ i < length a ∧ 0 ≤ j < length a → [a[i]/it1,a[j]/it2]exp
a→**exists**(it: DT | exp)	∃ i: int. 0 ≤ i < length a ∧ [a[i]/it]exp
a→**exists**(it1, it2: DT | exp)	∃ i j: int. 0 ≤ i < length a ∧ 0 ≤ j < length a ∧ [a[i]/it1,a[j]/it2]exp
a→**one**(it: DT | exp)	∃ i: int. 0 ≤ i < length a ∧ $[a[i]/it]exp$ ∧ (∀ j: int. 0 ≤ j < length a ∧ j <> i → $[a[i]/it]exp$ <> $[a[j]/it]exp$)
a→**isUnique**(it: DT | exp)	∀ i,j : int. 0 ≤ i < length a ∧ 0 ≤ j < length a ∧ i <> j → $[a[i]/it]exp$ <> $[a[j]/it]exp$

Table 1. OCL logical property verification operations mapping to WHY expressions

3.6 Value extraction iteration operations

Iteration operations also allow extracting values from a collection: according to the satisfaction (select) or not (reject) of a property; or by applying a treatment on every element of a list (collect). Value extraction operations are more complex to model as they do not only provide a single boolean value as output, they actually compute a list of elements.

Iteration operations semantics Iteration operations apply on collections and compute filtering of the collection values and/or mapping of functions on the collection values. We decide thus to represent a collection as the $\langle c, p, f\rangle$ tuple. Its semantics is provided in (1).

$$[\![\langle c, p, f\rangle]\!] = \{f(v)|v \in c, p(v)\} \tag{1}$$

According to the previous notation, we define the initial value of a collection as in (2) where $\top$ is the predicate returning *true* and `id` the identity relation.

$$c = \{v_1, ..., v_n\} = \langle c, \top, \mathtt{id}\rangle \tag{2}$$

The definition of iteration operations is hence specified as in (3). From these definitions, we extract the implementations for iteration operations using the WHY language. We provide in the following two possible implementations.

$$\begin{array}{rrr}
\langle c, p, f\rangle \rightarrow & \mathit{select}(e|\varphi) = & \langle c, p \wedge [f/e]\varphi, f\rangle \\
\langle c, p, f\rangle \rightarrow & \mathit{reject}(e|\varphi) = & \langle c, p \wedge \neg[f/e]\varphi, f\rangle \\
\langle c, p, f\rangle \rightarrow & \mathit{any}(e|\varphi) = & \langle c, p, f\rangle \rightarrow \mathit{select}(e|\varphi) \rightarrow \mathit{first}() \\
\langle c, p, f\rangle \rightarrow & \mathit{collect}(g) = & \langle c, p, f \circ g\rangle
\end{array} \tag{3}$$

Iteration operations as first order logic constructs Providing an implementation for iteration operations can be done using first order logic constructs. It is then required to generate a different function for each iteration operation call and then to write a call to the generated function in the translated OCL operation code.

Using first order logic to provide an implementation for iteration operations is quite straightforward. Nevertheless, the implementation and verification of the generation process might be quite complex. Indeed the generated code complexity might increase when complex expressions are used in the body of the iteration operation.

Iteration operations as higher order logic construct An alternative approach for the implementation of iteration operations is to rely on higher order logic to represent the operations in WHY. An example of such formalisation is provided in Listing 1.1 for the *select* iteration operation. The *select* function in WHY takes two arguments, the first one is the collection on which the operation is applied and the second one is the predicate that must be used in order to select the elements of the collection. In addition to the function definition, we provide a set of lemmas verified with the WHY3 platform generating proof obligations discharged using SMT solvers and proof assistants. In the case of the *select* function, part of the lemmas are verified using SMT solvers and others have been verified using the COQ proof assistant. We provide in Figure 2 a report generated with the WHY3 toolset for these verifications. The verification effort required for the verification of the iteration operations (writing of the specification and achieving the proof) is rather similar to the one needed for the verification of collection operations. Details regarding the formalisation and the proofs for the other OCL collection operations is provided in the first author PhD thesis work [5], and our website[6].

[5] http://www.dieumegard.net/thesis/Thesis-Arnaud_Dieumegard.pdf
[6] http://block-library.enseeiht.fr

In order to use the iteration expressions, one must provide a body containing the condition or the operation to apply on each element of the collection. The condition body of the *reject*, *select* and *anyAs* operations is translated as an inlined predicate whereas the function body of the *collect* operation is translated as an in-lined function. In-lined predicates and functions are provided as the second argument of their respective WhyML function call. Contrary to the first order operations, there is no need here to keep track of the variables used in the iteration operation as the in-lined nature of the function call makes their definition directly available.

```
function select (l: list oclType) (p: HO.pred oclType) : list oclType =
    match l with
    | Nil -> Nil
    | Cons hd tl -> if p hd then Cons hd (select tl p)
                            else select tl p
    end

lemma select_nil: forall p: HO.pred oclType.
  select Nil p = Nil
lemma select_cons_nil_verified: forall e: oclType, p: HO.pred oclType.
  p e -> select (Cons e Nil) p = Cons e Nil
lemma select_cons_nil_not_verified: forall e: oclType, p: HO.pred oclType.
  not (p e) -> select (Cons e Nil) p = Nil
lemma select_cons_verified: forall e: oclType, l: list oclType,
                                   p: HO.pred oclType.
  p e -> select (Cons e l) p = Cons e (select l p)
lemma select_cons_not_verified: forall e: oclType, l: list oclType,
                                       p: HO.pred oclType.
  not (p e) -> select (Cons e l) p = select l p
lemma select_mem_reduc: forall l: list oclType, b: oclType,
                               p: HO.pred oclType.
  mem b (select l p) -> mem b l
lemma select_mem: forall l: list oclType, b: oclType,
                         p: HO.pred oclType.
  (mem b l /\ p b) -> mem b (select l p)
lemma select_not_mem: forall l: list oclType, b: oclType,
                             p: HO.pred oclType.
  (mem b l /\ not (p b)) -> not (mem b (select l p))
```

Listing 1.1. Select iteration operation formalisation in Why using higher order logic

Proof obligations	**Alt-Ergo-Pro (1.0.0)**	**Coq (8.4pl3)**
lemma `select_nil`	0.03	
lemma `select_cons_nil_verified`	0.04	
lemma `select_cons_nil_not_verified`	0.05	
lemma `select_cons_verified`	0.03	
lemma `select_cons_not_verified`	0.04	
lemma `select_mem_reduc`		2.40
lemma `select_mem`		2.31
lemma `select_not_mem`		2.01

Fig. 2. Select verification through SMT solvers and proof assistants (time in seconds)

3.7 Additional restrictions on the support of the OCL language

In the previous sections, we gave the translation rules for a subset of the OCL language. OCL provides a well known syntax for software engineers. In addition, this subset of the language eases and secures the constraints writing process relying on first order logic by including only well known and standard functions and operations on classical data types.

A major restriction in our implementation of the OCL language is the absence of the specific *invalid* and *null* values. This is related to the binding to the data part of the DSML and to the formal verification introduced by the WHY3 toolset. The *invalid* value is used for the modeling the failure of an operation, we advocate that if such an operation is equipped with pre-conditions then the *invalid* value can never be an output of the operation and is thus not necessary. This value is also used when OCL is bound to UML or MOF to model optional attributes or references that are not defined in the model instances. We advocate that an empty collection is also a good model for this case. Regarding the *null* value, it is used in order to model an object that does not exists. Once again this should not happen in a formalisation of the language but it could be handled by relying on an *option* type. Adding the support for these specific values will have a strong impact on the WHY implementation of the OCL constructs. Indeed, these specific values will need to be taken into account in the implementation of all functions and on the related proofs. There are technical difficulties in implementing these values as shown by [2] but it is possible to overcome most of them.

The limitations we applied on the formalisation of OCL have the advantage of providing a simpler, well defined subset of the language enforcing the DSML developer to rely mostly on common knowledge OCL constructs and thus avoid the problems related to the use of more complex constructs such as the *closure* operator. These operators may cause some misunderstandings; make the specification more obscure; and make its verification more complex.

The subset of OCL we currently handle could be extended if required by the integration to other DSMLs as shown by [2]. We will try to keep it as limited as possible.

4 Related Works

Many works in the literature target the formal verification of DSMLs. We will only focus on those where the OCL language is used either in the DSML specification and/or as a language component part of the DSML; and is included in the verification process. Most of the formal DSML verification processes including OCL are related to its use in the context of UML specifications.

Our approach is close to the one proposed in the HOL/OCL project[3, 2]. Its authors target the formal verification of UML specifications including OCL constraints and contracts by providing a bridge to the HOL/OCL whose formal foundations relies on the ISABELLE proof environment. This work is specifically remarkable by its coverage of OCL and has highly contributed to the OCL community by providing formal specifications for the language that raised numerous legitimate questions regarding the semantics of OCL operations and constructs leading to the clarification of the standard. Our work takes a different angle and aims at the verification of DSMLs in general whereas the HOL/OCL one is focused on UML. We also differ by our use of the WHY3 toolset instead of ISABELLE. Both toolsets allows for the use of SMT solvers for the automatic verification of properties but WHY3 can be used as a bridge to many more different solvers and proof assistants including ISABELLE. WHY3 also benefits from its simpler high level syntax. It is our belief that by relying on the WHY3 toolset, a wider range of users will be able to actually perform formal verifications of properties as the WHY and WHYML languages are simpler to apprehend than ISABELLE. Our work targets a significantly smaller coverage of the OCL standard that allows avoiding key semantics issues like the *invalid* and *null* values. This has not been a limiting issue for our use up to now. However, we are far from the completeness of the HOL/OCL project and we do not plan to reach it.

Clavel et al [4, 7] provides a translation from OCL to first order logic (FOL). Automated theorem provers and SMT solvers are then used to check the unsatisfiability of the generated FOL constructs. This work supports the *null* and *invalid* values. It focuses on the verification of OCL constraints expressed on data models. Our work is more focused on the verification of OCL as a language component for DSMLs. The same difference can be found in the work by Lano et al [15] where UML class diagrams and a subset notational variant of OCL (LOCA) are handled and verified through a transformation to the B language.

UML/OCL specifications are translated to SAT solving format by Soeken et al [18, 19] or integrated in the USE toolset by Kuhlmann et al [12]. Another formalism for the automated verification of specifications is the PVS theorem prover. Kyas et al in [13] provides a lightweight translation of OCL and UML state machines and class diagrams. Each of these approaches uses different formalism for the formal verification with efficient verification results but are tied up to the use of this specific formal verification technique. Our work benefits from the wider range of formal verification toolsets that the WHY3 platform provides including most of the ones used in these works.

Some other approaches have been used in order to formally model UML/OCL specifications as in the works from Kyriakos et al [1] and Cunha et Al[6] on the UML2Alloy tool. These provide means for the verification of UML/OCL specifications and put a specific emphasis on the feedback of the verification result to the DSML user. We did not focus on the feedback from solvers in our work but it would be possible as the WHY3 toolset provides access to SMT solvers with counter-example generation capabilities.

5 Conclusion and Future Works

We have described a lightweight formalisation of the OCL language in WHY3 where the limitations have been identified and discussed. We have briefly detailed an example of DSML where OCL is used as a component language. We rely on our formalisation of OCL to conduct the automated verification of high level properties on the DSML instances.

It is our belief that the DSML community would benefit from such a formalisation as it allows to improve the quality of model verifications by relying on formal verification techniques; it is based on the use of the WHY3 toolset that eases the writing of properties and is more accessible for industrial practitioners than other formal verification platforms (like proof assistants for example) yet still supporting their use. This accessibility is very likely to help on the adoption of formal methods as a natural and standard verification technique.

The work conducted here was mostly driven by the verification of a specific DSML: the BLOCKLIBRARY specification language. The next step is to ease the integration of OCL as a component language for other DSMLs. Specifically, we will focus on the adaptation of the translation of the DSML structure to WHY3 which is currently done manually. We plan on providing a translation from meta-models instances of MOF or ECORE to generically transform any DSML specification into a set of WHY theories. The translated theories will then be used in conjunction with our formalisation of OCL to conduct verification activities on the DSML instances.

We are investigating a larger subset of OCL that still allows for an efficient automatic verification of DSML properties. We will specifically focus on the handling of any type of collections; the constructs that still need handling (tuples); and collection operations we did not implement yet. Handling the special *invalid* and *null* values could be done by relying on an approach close to the one provided by Dania et al in [7].

In our BLOCKLIBRARY DSML the blocks semantics is specified using an action language. Such a semantics is then translated as a WHYML function. We could have used an already existing action language like FUML/ALF [17] for example. By relying on a custom language, we simplified the translation work by limiting it only to the constructs that were needed. We plan, as proposed for a subset of OCL, on providing a formalisation for a subset of such an action language component. This will then be used for the verification of the semantics definition for DSMLs and will thus complete the DSML verification approach.

References

1. Kyriakos Anastasakis, Behzad Bordbar, Geri Georg, and Indrakshi Ray. On challenges of model transformation from UML to alloy. *Software & Systems Modeling*, 9(1):69–86, 2010.
2. Achim D. Brucker, Frédéric Tuong, and Burkhart Wolff. Featherweight OCL: A proposal for a machine-checked formal semantics for OCL 2.5. *Archive of Formal Proofs*, January 2014.
3. Achim D. Brucker and Burkhart Wolff. HOL-OCL: A formal proof environment for UML/OCL. In *FASE*, volume 4961 of *LNCS*, pages 97–100. Springer, 2008.
4. Manuel Clavel, Marina Egea, and Miguel Angel García de Dios. Checking unsatisfiability for OCL constraints. *ECEASST*, 24, 2009.
5. Benoit Combemale, Xavier Crégut, and Marc Pantel. A Design Pattern to Build Executable DSMLs and associated V&V tools. In *The 19th Asia-Pacific Software Engineering Conference (APSEC 2012)*, Hong Kong, Hong Kong, December 2012. IEEE.

6. Alcino Cunha, Ana Garis, and Daniel Riesco. Translating between alloy specifications and UML class diagrams annotated with OCL. *Software & Systems Modeling*, 14(1):5–25, 2015.
7. Carolina Dania and Manuel Clavel. OCL2FOL+: coping with undefinedness. In *Proceedings of the MODELS 2013 OCL Workshop co-located with the 16th International ACM/IEEE Conference on Model Driven Engineering Languages and Systems (MODELS 2013), Miami, USA, September 30, 2013.*, pages 53–62, 2013.
8. Julien Deantoni and Frédéric Mallet. ECL: the Event Constraint Language, an Extension of OCL with Events. Research Report RR-8031, July 2012.
9. Arnaud Dieumegard, Andres Toom, and Marc Pantel. Model-based formal specification of a DSL library for a qualified code generator. In *Proceedings of the 12th Workshop on OCL and Textual Modelling*, pages 61–62, New York, NY, USA, 2012. ACM.
10. Arnaud Dieumegard, Andres Toom, and Marc Pantel. A software product line approach for semantic specification of block libraries in dataflow languages. In *18th International Software Product Line Conference, SPLC '14, Florence, Italy, September 15-19, 2014*, pages 217–226. ACM, 2014.
11. Florian Heidenreich, Jendrik Johannes, Sven Karol, Mirko Seifert, Michael Thiele, Christian Wende, and Claas Wilke. Integrating OCL and textual modelling languages. In *Models in Software Engineering*, volume 6627 of *LNCS*, pages 349–363. Springer, 2011.
12. Mirco Kuhlmann, Lars Hamann, and Martin Gogolla. Extensive validation of OCL models by integrating SAT solving into USE. In *Objects, Models, Components, Patterns*, volume 6705 of *LNCS*, pages 290–306. Springer Berlin Heidelberg, 2011.
13. Marcel Kyas, Harald Fecher, Frank S. de Boer, Joost Jacob, Jozef Hooman, Mark van der Zwaag, Tamarah Arons, and Hillel Kugler. Formalizing UML models and OCL constraints in PVS. *Electr. Notes Theor. Comput. Sci.*, 115:39–47, 2005.
14. Christian Laasch and MarcH. Scholl. A functional object database language. In *Database Programming Languages (DBPL-4)*, Workshops in Computing, pages 136–156. Springer London, 1994.
15. K. Lano, D. Clark, and K. Androutsopoulos. Uml to b: Formal verification of object-oriented models. In *Integrated Formal Methods*, volume 2999 of *LNCS*, pages 187–206. Springer Berlin Heidelberg, 2004.
16. David Mentré, Claude Marché, Jean-Christophe Filliâtre, and Masashi Asuka. Discharging proof obligations from Atelier B using multiple automated provers. In *ABZ'2012 - 3rd International Conference on Abstract State Machines, Alloy, B and Z*, volume 7316 of *LNCS*, pages 238–251, Pisa, Italy, June 2012. Springer.
17. Isabelle Perseil. ALF formal. *Innovations in Systems and Software Engineering*, 7(4):325–326, 2011.
18. M. Soeken, R. Wille, M. Kuhlmann, M. Gogolla, and R. Drechsler. Verifying UML/OCL models using boolean satisfiability. In *Design, Automation Test in Europe Conference Exhibition (DATE), 2010*, pages 1341–1344, March 2010.
19. Mathias Soeken, Robert Wille, and Rolf Drechsler. Encoding OCL data types for sat-based verification of UML/OCL models. pages 152–170, 2011.
20. Faiez Zalila, Xavier Crégut, and Marc Pantel. Leveraging formal verification tools for DSML users: A process modeling case study. In *Leveraging Applications of Formal Methods, Verification and Validation. Applications and Case Studies*, volume 7610 of *LNCS*, pages 329–343. Springer Berlin Heidelberg, 2012.
21. Faiez Zalila, Xavier Crégut, and Marc Pantel. Formal verification integration approach for DSML. In *Model-Driven Engineering Languages and Systems*, volume 8107 of *LNCS*, pages 336–351. Springer Berlin Heidelberg, 2013.

Tool Paper: Combining Alf and UML in Modeling Tools – An Example with Papyrus –

Ed Seidewitz

Model Driven Solutions
14000 Gulliver's Trail
Bowie MD 20720 USA
`ed-s@modeldriven.com`

Jérémie Tatibouet

CEA, LIST, Laboratory of Model Driven Engineering for Embedded Systems
P.C. 174, Gif-sur-Yvette, 91191, France
`jeremie.tatibouet@cea.fr`

Abstract. The Unified Modeling Language has been used largely in the software community to draw pictures for designing and documenting software written in other languages. The real executable semantics of a program are determined by the programming language, while the UML models themselves do not have a precise enough meaning to fully specify the executable functionality of the system being developed. Recently, however, there has been a great deal of work toward the standardization of precise, executable semantics for UML models – the "meaning" behind the pictures: Foundational UML (fUML) adopted by the Object Management Group in 2008, the Action Language for fUML (Alf) adopted in 2010, the recently completed Precise Semantics for UML Composite Structures (PSCS) and the Precise Semantics for UML State Machines (PSSM), now in progress. Together, these standards effectively provide a new combined graphical and textual language for precise, executable modeling. In particular, the Alf language goes beyond simply providing a textual "action language" for detailed behavioral code within graphical models, by including textual notation for fUML structural object-oriented modeling constructs (e.g., packages, classes, associations, etc.). This opens up the possibility of tooling allowing various parts of a UML model to be represented both graphically and textually (while preserving the same semantic level), with bidirectional synchronization between the two representations. This paper presents the achievement of an initial integration of UML and Alf in the context of the Papyrus tool for the specification of executable models.

Keywords. Graphical modeling. Textual modeling. UML. Alf. Action language. Modeling tools. Graphical/textual model synchronization.

1 Introduction

To most in the software community, “modeling” is something much different than “coding”. On the other hand, there has also been a long standing minority in the software development community that has created models precise enough that they can be executed in their own right. Indeed, commercial executable modeling tools based on the Shlaer-Mellor method [5,6], Real-Time Object-Oriented Modeling (ROOM) [4] and Harel statecharts [2] all predated UML. However, such approaches converted over to UML notations,[1] and executable UML has been used for significant and critical applications, including fighter aircraft flight software, launch vehicle flight software and telecommunication switches [1,7].

Nevertheless, executable modeling has remained a niche approach dependent on divergent, proprietary tooling. One crucial issue with creating precise, standard UML models has been the imprecision of semantics specification in the UML standard. This issue was finally addressed with the adoption by OMG of the Foundational UML (fUML) specification[2]. This specification provides the first precise operational and axiomatic semantics for a Turing complete, executable subset of UML. The subset encompasses most of the object-oriented and activity/action modeling constructs of UML, which cover not only features commonly found in an object-oriented programming language, but also more advanced modeling features found in UML such as first-class associations and asynchronous signals.

But there has been a second crucial issue with executable UML modeling: the lack of a good surface notation for specifying detailed behavior and computation. UML is a largely graphical modeling language whose legacy is the unification of earlier graphical modeling languages. This is a great strength of UML for traditional, largely informal “big picture” analysis and design modeling, but it does not work well for representing detailed computations.

The fUML specification does not provide any new concrete surface syntax, tying the precise semantics solely to the existing abstract syntax model of UML. UML does provide a concrete notation for activities and actions that can be used to model, say, the method for an operation, but this requires one to draw a very detailed, graphical activity diagram.

This issue was addressed with the adoption by OMG of the Action Language for fUML (Alf)[3]. Alf is basically a textual notation for UML behaviors that can be attached to a UML model anyplace that a UML behavior can be. Together, these standards effectively provide a new combined graphical and textual language for precise, executable modeling.

Such a combination of graphical and textual notation is being implemented in practice in the Eclipse-based open-source UML/SysML modeling tool Papyrus[4]. In addition to the usual diagrams, the tool now provides the user with a textual editor sup-

[1] See, for example, http://www.kc.com/XUML/ and http://www.xtuml.org/.
[2] http://www.omg.org/spec/FUML/
[3] http://www.omg.org/spec/ALF
[4] https://eclipse.org/papyrus/

porting Alf. When developing an executable model, one can easily switch between the different editing views. The cohesion between the different views is ensured in a transparent way for the user.

This paper has two objectives. The first one is to introduce the reader to Alf both from syntactic and semantic standpoints. The second objective is to demonstrate the coupling between Alf and UML through an example built using the tooling integrated into Papyrus.

The remainder of this paper is organized as follows. Section 2 provides some additional background on Alf as a textual modeling language. Section 3 then introduces a simple example UML model, and Section 4 shows how this model can be updated with executable Alf code using Papyrus. Section 5 then makes some additional points about the synchronization of model changes occurring in different views. Section 6 identifies the limitations of the current Alf tooling and Section 7 concludes the paper.

2 Background

Semantically, Alf maps to the fUML subset. In this regard, one can think of fUML as effectively providing the "virtual machine" for the execution of the Alf language. However, this grounding in fUML also provides for seamless semantic integration with larger graphical UML models in which Alf text may be embedded. This avoids the semantic dissonance and non-standard conventions required if one where to instead, say, use a programming language like Java or C++ as a detailed action language within the context of an overall UML model.

However, the Alf language actually also includes a notation that goes beyond just behavioral modeling constructs. This additional textual notation includes all the *structural* modeling constructs included in the fUML subset. For example, suppose we have a UML class model that has an association between a `Customer` class and an `Account` class. This simple model can be represented textually in Alf:

```
package CustomerAccounts {
  public class Customer {
    public name : String;
  }

  public class Account {
    public balance : Integer;
  }

  public assoc CustomerAccount {
    public customer : Customer;
    public accounts : Account[*];
  }
}
```

Now, given a certain customer, we want to navigate across the association to the sum up the balances of all the customer's accounts. We can use Alf to define a UML activity to do this:

```
private import CustomerAccounts;
activity SumBalances(in customer : Customer) : Integer {
  totalBalance = 0;
  for (balance in customer.accounts.balance) {
      totalBalance += balance;
  }
  return totalBalance;
}
```

Syntactically, Alf looks at first much like a typical C/C++/Java legacy language. This is the result of a conscious compromise on the part of the submission team. Since, despite the issues involved, it is currently not uncommon practice to use Java or C++ as a UML action language, there was a strong desire to have a subset of Alf that would be familiar to such practitioners, to ease their transition to the new action language.

But the notational similarity can also be a bit deceptive. For example, association ends in UML are not semantically collection objects, but, rather, multi-valued properties with specified multiplicities (such as `[*]` used above, meaning "0 to many"). So, while `customer.accounts.balance` may look like a regular Java field access expression, what it really does is navigate from `cutomer`, across the association to the opposite **`accounts`** end, return all the **`Account`** objects at that end, and get the **`balance`** of each one. Alf adopts the notational convenience introduced in the already standard Object Constraint Language (OCL)[5] that navigation across an association to a multi-valued end automatically collects all the values at that end, so it is not necessary to have an explicit for loop to do this.

Note also that it is not necessary to explicitly declare the type of `totalBalance` or `balance`. The types of these local names are inferred from the result types of the expressions assigned to them – a convenience familiar to users of any modern scripting language.

Further, beyond simple syntactic conveniences, Alf also includes constructs that leverage the inherently concurrent, flow-oriented nature of the underlying fUML activity semantics. These include very powerful capabilities like filtering and mapping similar to those seen in many of the recently popular functional languages. So, for example, the body of `SumBalances` above can be more compactly written as:

```
return customer.accounts.balance->reduce '+';
```

Here, the functionality of an entire loop has been collapsed into a single expression, which maps directly to a UML reduce action. On an appropriate platform, this could be implemented as a highly concurrent operation, rather than as a sequential loop.

5 http://www.omg.org/spec/OCL/

Further, suppose that the customer was to be selected based on email address from the extent of existing customers. This can be simply written:

```
myCustomer =
  Customer->select c (c.email == myCustomerEmail);
total = SumBalances(myCustomer);
```

The `select` notation here maps to a fUML parallel expansion region that, again, could be implemented as a highly concurrent search – or even translated into a database query. And, while the = looks like a traditional variable assignment, what it really maps to is a data flow in the underlying fUML activity – so local assignments do not actually introduce mutable state, which again allows much greater flexibility in the translation to implementation.

The point is that Alf provides an essentially complete notation for writing programs at the level of UML modeling semantics. Indeed, the open-source Alf Reference Implementation[6] is distributed un-integrated with any graphical tool, allowing executable models to be written completely textually in Alf in exactly this way. Nevertheless, a particular benefit of Alf is its close relationship to standard UML, which allows it to be integrated readily as a textual notation into existing graphical UML tools. But the availability of the extended Alf notation for structural modeling opens up the possibility of integration beyond just including Alf snippets for behavioral functionality within a graphical UML model. We turn next to consideration of tooling that realizes this possibility.

3 Papyrus support for Alf

Implementation of Alf in the context of Papyrus is the result of a collaboration between Model Driven Solutions (language implementation) and CEA LIST (integration in Papyrus). As shown in **Fig. 1**, this implementation is structured as a number of Eclipse plugins, which can be grouped into two parts, a “back end” and a “front end”.

6 http://alf.modeldriven.org

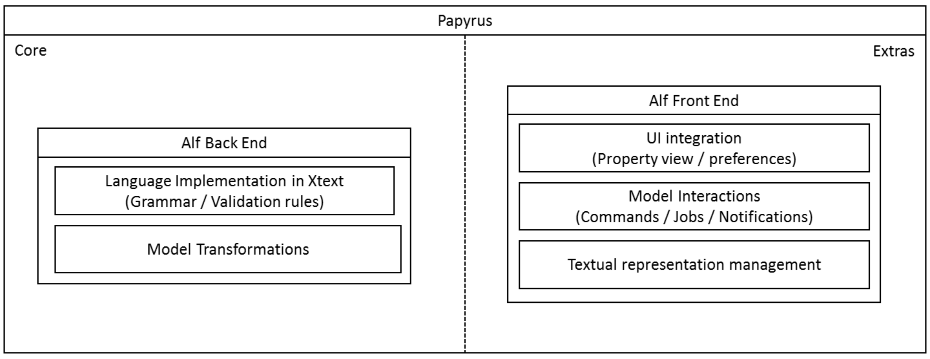

Papyrus
Core
Extras
Alf Back End
Language Implementation in Xtext
(Grammar / Validation rules)
Model Transformations
Alf Front End
UI integration
(Property view / preferences)
Model Interactions
(Commands / Jobs / Notifications)
Textual representation management

Fig. 1. Alf support architecture in Papyrus

The "back end" provides a complete implementation of the Alf language. The syntax of the language is defined as an Xtext[7] grammar. This grammar is used to parse Alf text into an Ecore metamodel based on the normative Alf abstract syntax. Semantic validation rules are given as OCL constraints that annotate this metamodel, which are executed during automatic validation as part of the Xtext framework. In addition, the "back end" also implements the mapping of the Alf abstract syntax to the UML abstract syntax using QVTo[8] transformations. Finally, a reverse QVTo transformation is also provided for mapping UML to the Alf abstract syntax, which may be then serialized to Alf text, in order to allow bidirectional synchronization between Alf and UML representations.

The "front end" enables the end user to use the Alf language implementation in the context of a UML model designed in Papyrus. It contributes to the property view and proposes an additional tab "ALF" (see, for example, **Fig. 3**) containing the required elements to specify and propagate Alf specifications entered by user in an existing UML model. The "front-end" handles all the interactions with the edited model through a set of commands and jobs. Additionally it provides experimental developments to maintain a synchronization between graphical and textual views of a single model.

The next section illustrates the use of Alf tooling integrated into Papyrus through an example.

4 A Simple (but Representative) Example

To show how graphical/textual model integration can work in UML tooling, it is easiest to use an example. We will use an example from the domain of e-commerce, a simple model of an order. The first thing to understand is what information needs to be kept on an order and how this is related to information on the customer placing the

[7] https://eclipse.org/Xtext/documentation/
[8] https://projects.eclipse.org/projects/modeling.mmt.qvt-oml

order. This can be well represented using a UML class diagram, such as the one shown in **Fig. 2**.

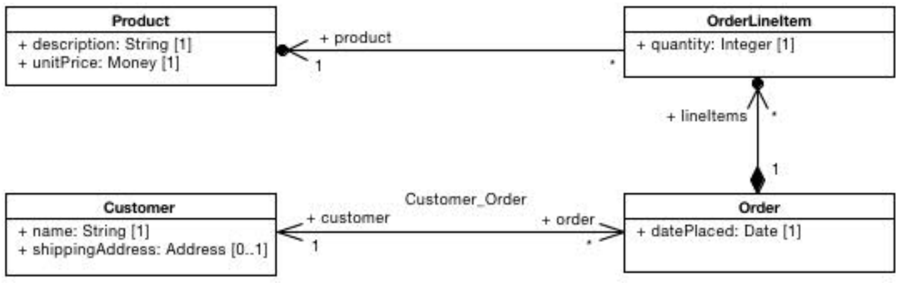


Fig. 2. Order example class diagram

This diagram was entered graphically using Papyrus. It models an order as recording the date it was placed and having a set of line items, each of which specifies the quantity of a certain product included in the order. It also shows that an order is placed by a single customer, who may have many orders.

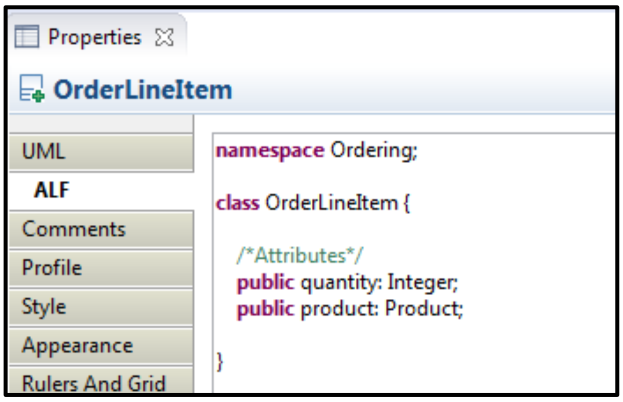
Models such as this are particularly useful in discussions with problem domain stakeholders. They are straightforward to understand and a lot of detail can be presented in a well-laid-out, compact diagram. For most people, this is far easier to understand than large blocks of text or written descriptions such as the previous paragraph.

Of course, there is also behavior associated with the classes shown in the diagram. Suppose, for instance, that you would like to add a `totalAmount` attribute to the `Order` class, along with an `addProduct` operation that adds a new line item for a given product and updates the `totalAmount` appropriately. To do this, the `addProduct` will use a new `getAmount` operation on `OrderLineItem`.

Rather than doing this by adding elements in multiple steps on the diagram, one can often specify them more efficiently by just typing text. We will show next how this can be done in Papyrus.

5 Adding Alf Text

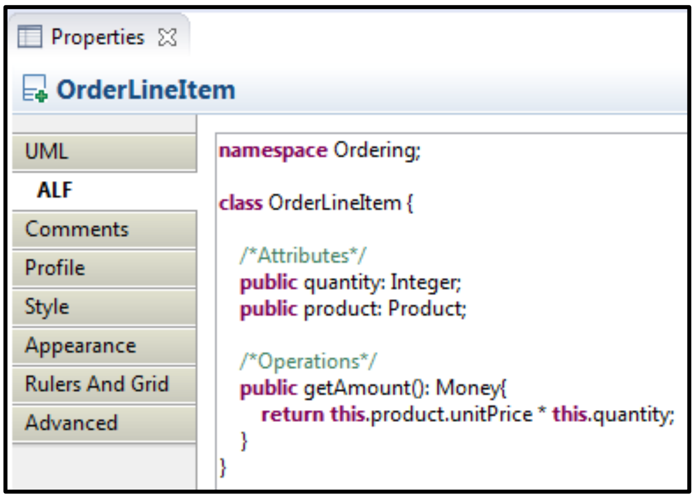
In the context of Papyrus, the Alf editor is only available when you select a model element that is in the scope of fUML (i.e., a Class, a Package, a Signal, an, Enumeration, a Datatype, an Association or an Activity). As an example, if you click on the `OrderLineItem` class either on the diagram (cf. **Fig. 2**) or in the model explorer, the textual specification corresponding to this element is rendered in the editor, as illustrated in **Fig. 3**.



```
namespace Ordering;

class OrderLineItem {

  /*Attributes*/
  public quantity: Integer;
  public product: Product;

}
```

Fig. 3. Alf specification of `OrderLineItem`

Now you can simply type the new `getAmount` operation directly into this textual representation, as shown in **Fig. 4**.



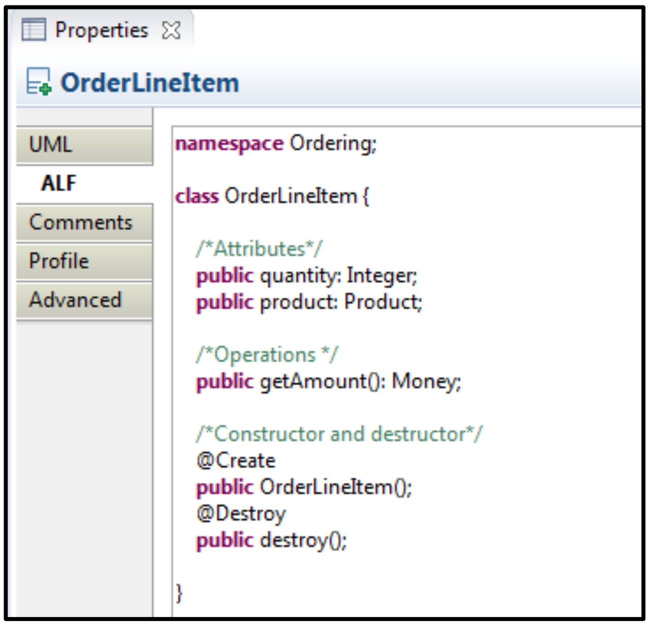
```
namespace Ordering;

class OrderLineItem {

  /*Attributes*/
  public quantity: Integer;
  public product: Product;

  /*Operations*/
  public getAmount(): Money{
    return this.product.unitPrice * this.quantity;
  }
}
```

Fig. 4. A new operation with its implementation

As soon as something new is added to the Alf specification, you can *compile*[9] it, in order to propagate the changes to the UML model. The compilation feature is only available if the model is correct both syntactically and semantically. The validation process is triggered each time a modification is made in the specification.

[9] The compilation is a user-triggered operation that starts when the user clicks on the "compile" button available below the Alf editor. The compilation consists in taking an Alf specification and building the corresponding fUML model.

In the current example, the result of the compilation is that a new operation is added to the class `OrderLineItem`. After compilation, the textual specification is updated, as shown in **Fig. 5**.


Properties

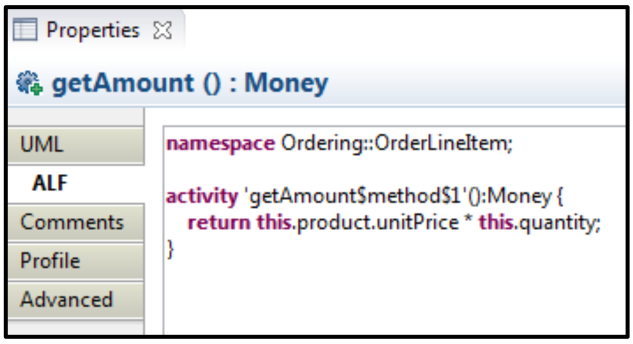
OrderLineItem

UML | ALF | Comments | Profile | Advanced

```
namespace Ordering;

class OrderLineItem {

  /*Attributes*/
  public quantity: Integer;
  public product: Product;

  /*Operations */
  public getAmount(): Money;

  /*Constructor and destructor*/
  @Create
  public OrderLineItem();
  @Destroy
  public destroy();

}
```

Fig. 5. Class `OrderLineItem` after compilation

Notice that the body of the operation `getAmount` no longer appears in the class definition. However, it has not really disappeared. As part of the compilation process, a UML activity was added to the model to hold the implementation of the operation. In UML terminology, this is known as the *method* of the operation. Clicking on this element in the model shows the textual representation in **Fig. 6**, which does, indeed, have the body of the operation, as originally entered.


Properties

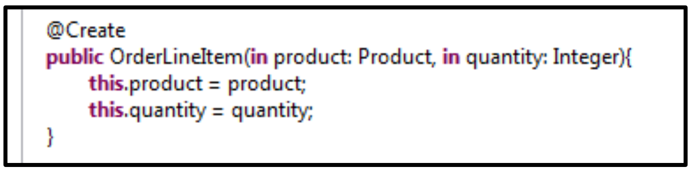
getAmount () : Money

UML | ALF | Comments | Profile | Advanced

```
namespace Ordering::OrderLineItem;

activity 'getAmount$method$1'():Money {
  return this.product.unitPrice * this.quantity;
}
```

Fig. 6. Implementation of the `getAmount` operation

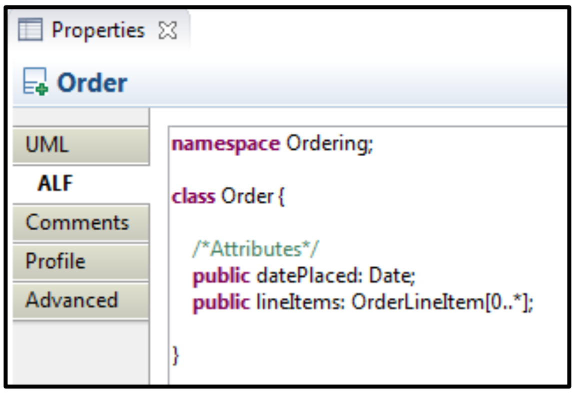
Note also that the compiler has automatically added default constructor and destructor operations to the `OrderLineItem` class, annotated with `Create` and `Destroy` stereotypes. In particular, you will need to use the `OrderLineItem` constructor to create a new `OrderLineItem` in the `addProduct` operation. And it would be more useful if the constructor was first updated to take line item `product` and `quantity` arguments, shown in **Fig. 7**.



```
@Create
public OrderLineItem(in product: Product, in quantity: Integer){
	this.product = product;
	this.quantity = quantity;
}
```

Fig. 7. `OrderLineItem` constructor implementation

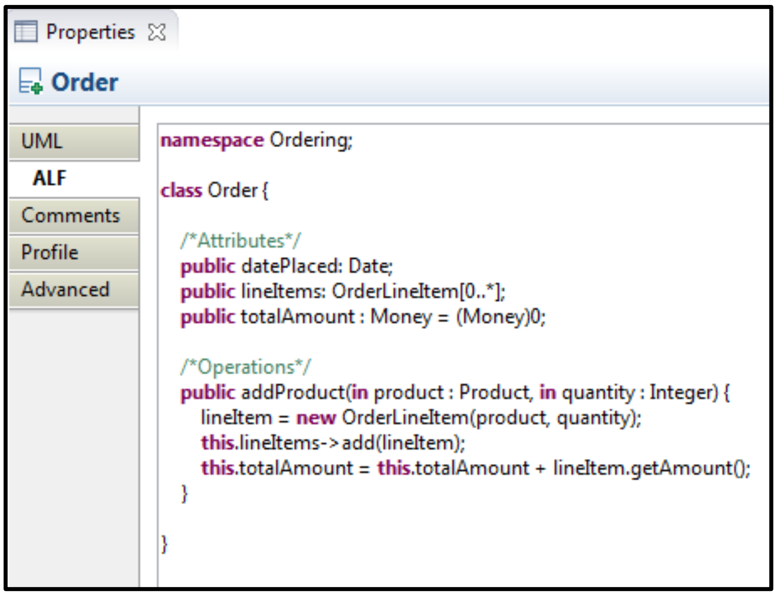
Now you are ready to add the new attribute and operation to `Order`. So, click on the `Order` class on the diagram, getting the textual representation shown in **Fig. 8**.



```
namespace Ordering;

class Order {

  /*Attributes*/
  public datePlaced: Date;
  public lineItems: OrderLineItem[0..*];

}
```

Fig. 8. The textual specification corresponding to the `Order` class

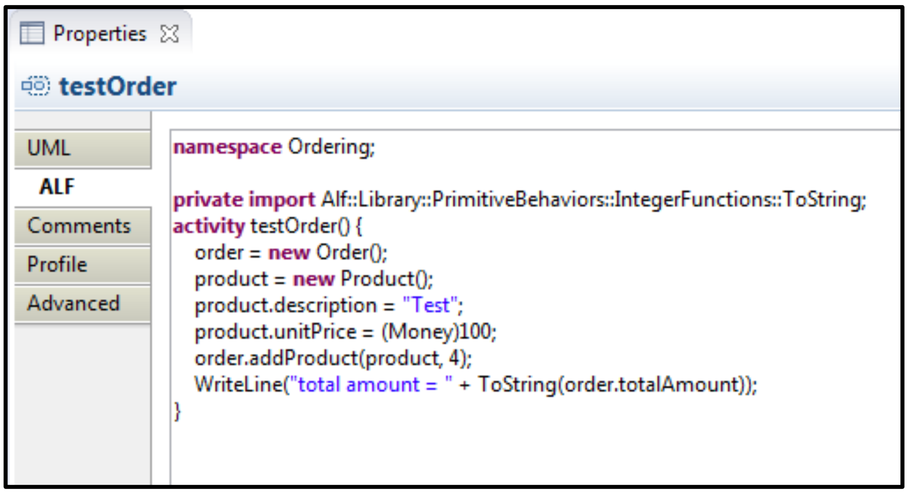
Then add the `totalAmount` attribute of type `Money` and the operation `addProduct`, with its implementation, as shown in **Fig. 9**. Finally, compile the result, in order to propagate your changes within the model.



```
namespace Ordering;

class Order {

   /*Attributes*/
   public datePlaced: Date;
   public lineItems: OrderLineItem[0..*];
   public totalAmount : Money = (Money)0;

   /*Operations*/
   public addProduct(in product : Product, in quantity : Integer) {
      lineItem = new OrderLineItem(product, quantity);
      this.lineItems->add(lineItem);
      this.totalAmount = this.totalAmount + lineItem.getAmount();
   }

}
```

Fig. 9. `Order` class updated with a new attribute and a new operation

We can now end this example by creating a little test program for our simple model, entering it as the UML activity shown in **Fig. 10**.



```
namespace Ordering;

private import Alf::Library::PrimitiveBehaviors::IntegerFunctions::ToString;
activity testOrder() {
   order = new Order();
   product = new Product();
   product.description = "Test";
   product.unitPrice = (Money)100;
   order.addProduct(product, 4);
   WriteLine("total amount = " + ToString(order.totalAmount));
}
```

Fig. 10. A test of the example model

Since Alf compiles to fUML, the compiled activity can be executed using Moka, the Papyrus model execution framework.[10] When interpreted, the model provides the expected result:

```
total amount = 400
```

This section presented a simple executable model combining UML notation and Alf. What we have seen is that textual representation introduces an additional view of the underlying model, by which the user can actually modify the model. Maintaining cohesion between multiple views (such as graphical and textual) and a model is a challenging task. The next section briefly discusses the existing support for multiple synchronized views, and their usefulness.

6 Keeping text and model synchronized

In the context of Papyrus two behaviors are supported in order to synchronize what is in the model with what appears in the graphical representation (i.e. the diagrams).

1. A model element can be partially synchronized with its graphical representation. For example if a class "A" containing two properties has a representation in a diagram, perhaps only one of these properties may be shown. If something changes about this property, its representation is of course updated. However if something else is added to the class, its representation remains the same.
2. Full synchronization can be maintained between the model and the graphical representation. This means that for a model element that appears on a diagram, any information about its owned elements is shown. **Fig. 11** illustrates the full synchronization between the `Order`[11] class and its view in the diagram. Properties operations, methods and other elements all appear in the graphical representation.

[10] https://wiki.eclipse.org/Papyrus/UserGuide/ModelExecution

[11] Note the naming convention used for methods of operations. These are introduced by the compilation process.

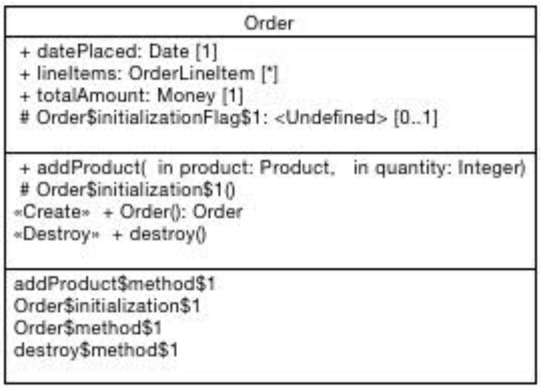

Order
+ datePlaced: Date [1]
+ lineItems: OrderLineItem [*]
+ totalAmount: Money [1]
# Order$initializationFlag$1: <Undefined> [0..1]
+ addProduct(in product: Product, in quantity: Integer)
# Order$initialization$1()
«Create» + Order(): Order
«Destroy» + destroy()
addProduct$method$1
Order$initialization$1
Order$method$1
destroy$method$1

Fig. 11. Full `Order` class graphical synchronization

The integration of the Alf tooling allows a user to edit the same model through two different views in Papyrus. The main problem here is to ensure that when a modification of the model is performed through a specific view, then other views are synchronized according to these changes.

In the context of the Alf tooling, we take advantage of EMF notifications to determine which textual representation have to be re-computed when the model changes. Say we have a class “B” that inherits from the class “A” and both are located in Package “SimplePackage”. If class “A” is renamed to “C”, then this implies the textual representation of B has to be computed as well has the one of “SimplePackage” that contains both classes.

There are, of course, cases in which the synchronization cannot be maintained between the views, and this is mainly related to the feature proposed by the tooling. In the context of Alf tooling, the user is allowed to modify the textual specification of a model element but is not forced to compile (i.e. to propagate the change to the model) right away. Indeed, the user has the ability to keep an incomplete or invalid Alf specification, which can be completed later on. However, this also introduce the possibility that the user may try to change the model from a different view, while ongoing changes remain pending for the textual view. This possibility is prevented by the tooling, which asks the user what to do. Typically, the user has the choice to keep or override his ongoing changes. Tooling relying on the Eclipse compare framework to enable the user to resolve manually a conflict, as it is proposed for example in Git, is expected in future developments.

The next section identifies current limitations of the Alf tooling.

7 Tooling limitations

The current version of the Alf tooling supports the basic features required to specify and modify an existing model using Alf. However, there is still a long way to go to bring this tooling to a level comparable to other existing professional programming environments. This section identifies three limitations that need to be addressed in future development to increase the level of maturity of our current tooling.

7.1 Auto-completion, cross liking and refactoring

The Alf editor needs to provide content completion, cross-linking and refactoring capabilities, just as expected in other Eclipse-based language editors. This will considerably accelerate the time required to obtain a valid specification.

7.2 Debugging

Papyrus provides the ability to execute fUML models, thanks to its model execution platform Moka. This also benefits from an integration with the Eclipse debugger, which makes it possible to interact with an execution and analyze manipulated values. The integration with the debugger does not yet allow the placement of breakpoints into Alf specifications and the propagation of debug information to the level of Alf source. Alf source-level debugging would make it a lot easier to debug complex behavior, which, when mapped to UML activity models, really look like compiled code.

7.3 Alf specification persistence

Although Alf specifications can be compiled into equivalent UML, it is usually desirable to persist the original Alf text for later retrieval. Indeed, it might not be possible to exactly reproduce the original text from the compiled models (e.g. user text formatting). Consequently, the text is for the moment stored in the model. The technical and standardized solution is to use a comment, stereotyped "TextualRepresentation" (with a tagged value "language = 'Alf'"), attached to the element mapped from Alf. The body of the comment contains the Alf code entered by the user.

The problem with this approach is that the model itself is modified to hold the persisted Alf text. A consequence of this is that, if the user tries to compare (for example with EMF Compare) the model with another version of the same model, there will be some differences (i.e., the comments containing the Alf text), which are not significant. Preliminary user feedback seems to indicate that it might be valuable in future versions of the Alf tooling to decouple the persistence of the UML model and the persistence of Alf textual specifications.

8 Conclusion

The Alf standard is new, and it will take some time for a new generation of tooling to be completed for it. But development of this tooling is progressing, with the vision of providing all the benefits of familiar IDEs for textual programming languages, along with the benefits of synchronized graphical views provided by a UML tool.

Unlike mainstream programming languages (e.g. Java, C++), Alf has been designed to smoothly integrate with UML, both syntactically and semantically. In this way, both the graphical and textual representations have the same precise, UML, model-level semantics.

In this paper we wanted to demonstrate that UML and Alf could in practice be used jointly to specify correct-by-construction and easily refinable system models. To do so, we presented in Section 3 a simple UML model built in Papyrus, which was then completed in Section 4 using Alf tooling we developed. The result was a model of an order system ready to be tested (i.e. executable). Beside the purely functional aspect of the tooling, note the flexibility brought by Alf in the specification of a model. Modeling choices can be updated smoothly in the model by a single click.

Although significant production applications have yet to be developed using the latest graphical and textual standards for executable UML modeling, it has already be used in significant research activities. Indeed an early prototype version of the Alf integration into Papyrus was used to specify the normative test suite for PSCS. And there is research [8, 9] into using fUML and Alf as the basis for specifying the semantics of domain-specific modeling languages.

We have also identified some limitations of the current Alf tooling compared to professional programming language environments, or based on user feedback. Removing these limitations will be an important focus in the future development of the technology.

References

1. Corcoran, D. 2011. *Model Driven Development and Executable UML at Ericsson.* http://www.omg.org/news/meetings/tc/agendas/va/xUML_pdf/Corcoran.pdf
2. Harel, D., and Politi, M. 1998. Modeling Reactive Systems with Statecharts: The Statement Approach. McGraw-Hill.
3. Mellor, S. J. and Balcer, M. J. 2002. Executable UML: A Foundation for Model-Driven Architecture. Addison-Wesley.
4. Selic, B., Gullekson, G. and Ward, P. 1994. *Real-Time Object-Oriented Modeling.* Wiley.
5. Shlaer, S. and Mellor, S. J. 1988.Object-Oriented Systems Analysis: Modeling the World in Data. Prentice Hall.
6. Shlaer, S. and Mellor, S. J. 1991. *Object Lifecycles: Modeling the World in States.* Prentice Hall.
7. Shubert, G. 2011. *Executable UML Information Day Panelist Presentation. Lockheed-Martin Space Systems Company.* http://www.omg.org/news/meetings/tc/agendas/va/xUML_pdf/Shubert.pdf
8. Tatibouet, J., Cuccuru, A., Gérard, S., and Terrier, F. 2014. Formalizing Execution Semantics of UML Profiles with fUML Models. MODELS.
9. Mayerhofer, T., Langer, P., Wimmer, M. 2013. xMOF: A Semantics Specification Language for Metamodeling. Satellite Events of MODELS.