Available online at www.sciencedirect.com
SCIENCE DIRECT
Electronic Notes in Theoretical Computer Science
ELSEVIER Electronic Notes in Theoretical Computer Science 102 (2004) 175–194
www.elsevier.com/locate/entcs

Combining OCL and Programming Languages for UML Model Processing

Mika Siikarla, Jari Peltonen, and Petri Selonen[1,2]

Institute of Software Systems
Tampere University of Technology
Tampere, Finland

Abstract

Model processing tasks, like model checking, merging, slicing, and synthesis, need efficient and maintainable mechanisms to define models, as well as to query, compare and manipulate information in them. Although the Object Constraint Language (OCL) is primarily meant for expressing constraints for UML models, it can also be used for various model processing purposes. In this paper we discuss the needs for, and possibilities of, using OCL for processing models, and show how we have applied and extended OCL. We also introduce a model processing tool using OCL as an integral part of model manipulation facilities. We especially emphasise the need of combined use of OCL and programming languages in UML model processing.

Keywords: OCL, UML, model processing, programming language, Python

1 Introduction

In UML-centered CASE-tools, the lack of rich support for different software engineering methodologies reflects the fact that UML [17] is a standard for a modeling language, not for a design method. This is both an opportunity and a challenge for the tool vendors. On one hand, there is a substantial freedom to implement any kind of method support to the tools. On the other hand, the users want to use and develop software processes that best fit their

[1] This work has been funded by the Academy of Finland and by Nokia. We would like to thank professor Tarja Systä for her valuable comments.
[2] Email: miksi@cs.tut.fi, jpe@cs.tut.fi, and pselonen@cs.tut.fi

doi:10.1016/j.entcs.2003.09.009

purposes. Accordingly, the tool vendors should be able to provide support for any methodologies the users want to use.

Our general research goal is to provide automated tool support for software engineering processes. Each of the processes consists of various tasks, like model checking, merging, slicing, synthesis, etc. requiring clear and maintainable mechanisms to efficiently define models, as well as query, compare and manipulate information in them. Additional functionality, required by process support, can be provided by using the application programming interfaces of the CASE-tools. These interfaces are typically used with a general purpose programming language (e.g. Rational Rose Basic [21]), allowing access to the model data.

When using a general purpose programming language for model processing, one quickly realises how ill-suited the tool is for the job. Simple queries often turn into multi-line nested loops or recursive calls, effectively hiding the user's intent. More complex searches manifest themselves in the code as long winded, difficult to understand functions. It does not take a big change in the query's matching criteria to force a rewrite of the entire section. Such changes are common during the early development and testing of a model processing task. The effects can be lessened by parametrising the query, as long as the changes concern only the values in the query, and not its structure. Also, the more complicated a piece of a program is, the harder it is to detect errors in it. There is a need for a high-level language for the UML domain.

The Object Constraint Language (OCL) [17, ch. 6] can rise to meet part of the requirements for a domain language. OCL is a formal language primarily meant for expressing constraints in UML models. The draft for the next version of OCL [3] extends the purpose to include all expressions on UML models, and specifically names queries as a possible use. Indeed, OCL is flexible and expressive enough for writing rather clear and complete queries and checks. But that is not all that is required for model processing. The OCL specification insists that the language is a specification language, not a programming language. There is no mention of how the operations should or could be implemented, or even if an implementation is possible or computationally feasible. It turns out, however, that it is easy to construct a naive algorithm for most, if not for all of the operations.

As a specification language OCL naturally lacks facilities for user interaction and for reading from, or writing to, files. But the biggest problem, from a model processing point of view, is that OCL is defined to be side effect free. It is therefore not possible to change a value of an object or create a link between objects. The restriction is an integral part of the language, affecting many of its aspects, and can not be simply ignored. Side effects can be described using

OCL, e.g. as post conditions, but in practise this approach is difficult to apply and does not yield easy to read operations. It would be possible to extend OCL to contain expressions ("old OCL", without side effects) and statements ("new OCL", with side effects). This is essentially the same as embedding OCL into another programming language, so we would be better off using an existing one. By mixing OCL expressions and a programming language for model processing, we get to use each of them for their intended purpose.

To summarise, we need a mechanism for processing UML models, providing not only primitive access to the model but also high-level support for implementing model processing tasks. In this paper we discuss the needs and possibilities of using OCL for model processing purposes, and show how we have applied and extended OCL. We also introduce a model processing environment, xUMLi [1], where we have used OCL as an integral part of model manipulation facilities. We especially emphasise the need of combined use of OCL and programming languages in UML model processing.

2 Model Processing

The main goal of our research is to provide automated tool support for various software engineering processes, each introducing a set of *model processing tasks*. We argue that these tasks can be performed by, and composed of, a set of primitive model processing operations. The *model operations*, in turn, are combined together to form a task using a higher-level composition mechanism, offering constructs such as flow of control (e.g. decisions, guards, flows, synchronization). We refer to the usage of these operations with the term *model processing*.

Perhaps the simplest example of a model operation is searching for, and filtering of, information in given UML models. These operations provide side effect free checking and validation of models. An obvious example of such an operation is enforcing the standard UML Well-Formedness rules [17]. Similarly, an operation could check whether a model follows given process or domain specific heuristics: for example, ensuring that all nodes in an inheritance tree should be abstract classes and all leaf classes should be concrete is a heuristics rule suggested by the OPEN software process [8, pp. 90]. Other examples of model operations are *transformation operations*, *projection operations*, *refactoring operations*, and *set operations*.

A *transformation operation* [25] takes a (set of) UML diagram(s) as its input operand and based on the information implied by this diagram produces a new UML diagram of another type. Examples of useful transformation operations include synthesis of statechart diagrams based on a set of interaction

diagrams [23][29][27], synthesis of structure diagrams from interaction diagrams [25][11], and synthesis of class diagrams from object diagrams [6]. A *projection operation* produces a new UML model based on an existing one, the new model being a projection of the original one (e.g. abstraction or slicing). As an example of a projection operation especially useful with large models produced during a reverse engineering process, consider generating compressed structure diagrams based on existing ones [20].

Refactoring operations produce new, modified models based on a given refactoring pattern; as an example, the Pull Up Method [7, pp. 332] states that if all subclasses of a given superclass have a method with identical results, this method is moved to the superclass. A *set operation* [24] (e.g. union, intersection, difference) takes two UML models as its input operands and produces a new UML model (e.g. merging and slicing diagrams). Set operations are particularly useful with processes (e.g. Catalysis [5]) and paradigms (e.g. Subject-Oriented Design [4]) that rely on mechanisms for composing specifications out of individual model fragments.

One particularly interesting category of model operations is *conformance operations*. These operations, enforcing conformance rules, are used for validating architecture design against domain or product-line specific profiles [26]. The architecture design is validated against the profiles using a configuration of suitable conformance rules. The rules can be seen as a set of OCL constraint templates that are instantiated by the profiles and then evaluated on the architecture views. The elements violating the conformance rules can be consequently collected and presented to the user. Currently, the techniques described in this paper, together with their implementation platform, are being evaluated with an industry study focusing on exploiting the conformance operations. The target system is a large-scale real-life model describing the architecture of one of Nokia's product lines.

The above mentioned model operations can also be seen to represent model transformations in the sense of Object Management Group's Model Driven Architecture (MDA) initiative [16]. In order to support model processing, feasible mechanisms are needed for querying, defining, and manipulating UML models. To implement such concepts in practice, one needs a high-level (i.e., UML domain specific) mechanism for implementing model operations for reasoning with UML models, an access to commercial UML CASE tools, and a way to combine the operations together.

3 Extending OCL - The Find Operation

OCL has a set of predefined types and operations for these types. They are defined in what is called the OCL Standard Library, and it is mandatory for implementations to include them. The predefined types include Integer, Real, String, Boolean, and a very useful set of structured data types. The type structure is also appended by all the classifiers from the current application model. In some situations there is need for more operations, e.g. taking the square root, or even new types. OCL offers powerful extension mechanisms for altering the language.

The lightest mechanism is the *let* expression, which allows the definition of a variable name and a corresponding expression. The name *variable* is a little misleading, since the variable name is textually replaced with the corresponding expression prior to evaluation. It is also possible to define variables that have arguments. *Let* resembles defining macros with the *#define* -directive in C and C++. There is also a stronger mechanism for attaching a variable to a type as a new property or a method. This is especially useful for augmenting a classifier from the application model with a method that simplifies the OCL expressions used. For serious extension needs, it is possible to define new basic types or new methods independent of the application model. These extensions are placed in a namespace and can be used in normal OCL expressions.

In the OCL expressions in this paper, we often use the *find* operation. It is not a part of the OCL standard library, but our extension to the language. The operation resembles *select* in that they both choose elements matching given criteria. However, in addition to checking the elements of the collection, *find* iterates over the children of each element recursively. That is, it descends down along any whole-part relations to the part and tests it, too. The search proceeds to the children of the tested part, and so on, until an element without children is reached. Such deep searches are required frequently, when operating on inherently hierarchical elements, such as *Packages* and *Namespaces* of the UML metamodel.

As an example of a *find* expression, let us consider the case of searching for new, user-defined stereotypes in a model. User-defined stereotypes offer a lightweight extension mechanism for UML, allowing the user to define new, domain specific concepts to be used in the models. As stated by [17], user-defined stereotypes are a means for refining the standard semantics of UML. They allow the modeler to add new modeling elements that can be used in UML models for process-specific or implementation language-specific domains. Let us assume that the stereotypes are located in a hierarchy of *Packages*, and that we are only interested in stereotypes under a certain *Package* in that

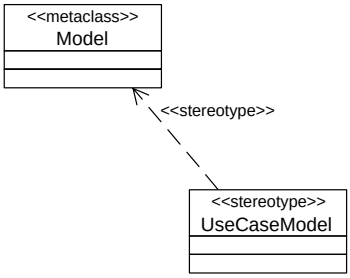

Fig. 1. Explicit stereotype modeling [17, Figure 4-1]

hierarchy.

We assume that the new stereotypes are modeled explicitly as defined in [17, pp. 4-3]. The original UML metaclass extended, is given as a *Class* marked with ≪metaclass≫, with the name of the class denoting the actual UML metaclass. The user-defined stereotype is given as a *Class* marked with ≪stereotype≫, its name being the name of the new stereotype. The latter *Class* is dependent on the former. The *Dependency* itself is marked with ≪stereotype≫. An example of such a stereotype definition is given in Figure 1, adopted from [17, Figure 4-1]. The diagram in Figure 1 introduces a new stereotype ≪UseCaseModel≫, which extends the standard UML metaclass *Model*.

The expression in Figure 2 recursively searches the *Package* hierarchy *self* for user-defined stereotypes. Let us take the hierarchy in Figure 3 of two *Packages*, five *Classes*, and one *Feature*. *Classes* S1 and S2 are user-defined stereotypes. The root *Package*, P1, contains *Package* P2, and *Classes* C1, S1, and C3. P2 contains *Classes* C2 and S2, and *Class* C1 contains one *Feature*, F1. *Package* knows its contents via the whole-part relationship *ownedElement* [17, Figure 2-32], and *Class* its *Features* through *feature* [17, Figure 2-5].

The first element checked is *self*, i.e. P1. Being a *Package*, not a *Class*, it fails the criteria. The search decends along *ownedElement* to P1's children. Similarly, it descends from P2 and C1 to their children. S1 and S2 match the criteria, and make up the resulting collection. If the same expression is applied to the *Package* P2, elements P2, C2, and S2 are checked and thus S2 is returned.

The existing extension mechanisms give the modeller powerful tools for enhancing and simplifying their expressions. Although new operations and even completely new kinds of types can be added to OCL, the mechanisms have their limitations. One can not change the underlying fundamentals of the language. It is not possible to make OCL expressions have side effect, or to dictate an algorithm to apply when a certain expression is evaluated, for

```
1.  self->find( element |
2.      element.oclAsType(Class).clientDependency->exists( cd |
3.          cd.stereotype.name->includes('stereotype')
4.          and cd.supplier.stereotype.name->includes('metaclass')
5.      )
6.  )
```

Fig. 2. Example of a Find operation

```
Package P1
    Class C1
        Feature F1
    Class S1 ≪stereotype≫
    Package P2
        Class C2
        Class S2 ≪stereotype≫
    Class C3
```

Fig. 3. A hierarchy of packages and classes

example.

4 Model Processing with Pure OCL

OCL is a specification language intended first and foremost for expressing constraints. It offers the basic functionality for performing calculations, comparisons, and string manipulation, as well as a useful collection of set operations. In general, the language is well-suited for expressing constraints. Constraints are just a form of checks, so it is not at all surprising that OCL is usable for checks, too. Searches were discussed, to an extent, in the previous section, and the conclusion is, that OCL has, or at least it can be extended to have, sufficient mechanisms for queries. Features required for a specification language are present, but that is not enough to make OCL by itself a suitable model processing language.

Since OCL is by definition side effect free, it can be used directly only for operations, which do not modify the state of the modelled system. This rules out OCL as a viable option for many model processing operations, e.g. the transformation operations. The restriction of not being able to modify the model is too inherent in the language to be dismissed lightly. There is more to changing that than simply introducing an assignment operator to the

```
Context Math::solveSecondDegree( a : Real, b : Real, c : Real ) : Real
Pre:    b*b - 4*a*c >= 0
Post:   a*result*result + b*result + c = 0
```

Fig. 4. An example of defining an operation using OCL post condition

syntax. If it was possible to modify objects, the evaluation of an expression could no longer be considered instantaneous. In order to keep the expressions deterministic, execution order would have to be defined. One might also have to rethink the way failed navigations and other operations are handled, etc. As stated before, the restriction is interwoven deep into the language. In order to change it, one would need to think carefully what other aspects might be affected.

Although the system can not be altered using OCL, expressions can be used to specify changes with, e.g. post conditions. The problems with this approach are mostly due to its impracticality. A post condition describes *what* the state of the system is like after the operation, not *how* to modify the existing state to achieve the result. Some kind of a mechanism for modifying arbitrary input into output matching a valid arbitrary OCL post condition would have to be developed. Even if it was possible in all cases, the execution of an operation would often be very inefficient without any hints of how to effectively achieve the result. The following simple example, although from the world of mathematics, illustrates this point. Let us consider the operation *solveSecondDegree* in Figure 4, which gives one real root of an equation of the form $ax^2 + bx + c = 0$. Without the knowledge of how to solve this particular type of equations, the mechanism would have to resort to more generic means, e.g. guesses of more or less educated kind.

It is possible to reduce the set of allowed OCL expressions in post conditions so, that the mysterious solving mechanism is always able to correctly construct the result. For example, we could allow only a single equation where the left hand side is a single object or navigation and the right hand side only refers to old values (or values that did not change). Actually, to make the post condition complete, we must also assume there is an implicit rule stating that "everything else in the system stays unmodified". An example of an operation with a restricted post condition is shown in Figure 5. Now, the right hand side of the expression can be evaluated normally and the value used as the new value of whatever was specified on the left hand side. The assignment takes place after the OCL expression is evaluated, and therefore we avoid the problems with side effects described earlier in this section.

In effect, we have introduced an assignment statement to the language by

```
Context Math::solveSecondDegree(a : Real, b : Real, c : Real ) : Real
Pre:    b*b − 4*a*c >= 0
Post:   result = ( −b + (b*b − 4*a*c).sqrt() ) / (2*a)
```

Fig. 5. An example of defining an operation using a restricted OCL post condition

separating the side effect free part from the assignment. The approach could be carefully modified to allow more than one post condition and even loops, but it does take some effort. Although it works with the simple example given, in practice operations are much more complicated and expressing them with above mentioned crippled post conditions yields hopelessly illegible operations. It takes a lot of skill and planning even for an implementer with extensive knowledge and experience on OCL to craft the necessary OCL expressions. If the operation is defined in a declarative way, translating it into a normal OCL post condition is often rather straight forward. This is no longer true for the limited post conditions. If the operation is defined as an algorithm, the translation becomes very difficult indeed.

These problems can be helped by separating side effect causing elements (statements) from side effect free ones (expressions). Expressions are standard OCL expressions, still instantaneous and completely side effect free. Statements are completely new language structures, which contain expressions and do have the ability to modify the model. Each expression is now a separate entity, and although the model does not change during the evaluation of a single expression, it can change after that expression, before another one is started. The previous post condition construction can now be honestly called assignment statement. While at it, we can add more statements, e.g. (control) *if* and *while*. This is essentially the same as defining a new programming language with embedded standard OCL. The same result could be achieved by taking an existing, mature, programming language and embedding OCL into that. All the existing features, e.g. interaction with the user, are instantly available to the programmer.

OCL is well suited for performing queries and checks on models. It is part of UML, and therefore naturally satisfies the requirement for a high level domain language. However, OCL is not a programming language, and should not be made one. By mixing a programming language and OCL, we get to use each tool for the purpose it was intended for as is best suited for. It is not even necessary to embed OCL into the language to get these benefits, as long as OCL is readily usable as expressions. OCL support could take the form of, e.g. a function library, or a program module.

```
1.  self.model->find( element |
2.      element.isKindOf(Class)
3.  ).stereotype.name->forAll( nm |
4.      self.profile->find( cl |
5.          cl.oclAsType(Class).clientDependency->exists( cd |
6.              cd.stereotype.name->includes('stereotype')
7.              and cd.supplier.stereotype.name->includes('metaclass')
8.          )
9.      ).name->includes( nm )
10. )
```

Fig. 6. Stereotype conformance predicate using OCL

5 Model Processing with a Programming Language

General purpose programming languages can, of course, be used for model processing. They often have a wide variety of structured data types, good selection of flow control statements, and sufficient mechanisms for user interaction, etc. However, for model processing purposes, they are a bit too general. Lacking direct support for processing UML models causes even simple queries to turn into long and complicated code. The point gets lost in the middle, and it is no longer clear what the high level idea was. This is the trade off between a general purpose programming language and a high level domain language. By narrowing the scope, a domain language can offer more complicated and better suited operations for specific tasks, but is no longer as useful for tasks outside its scope.

In this chapter we will use Python as an example, but the points apply to other programming languages as well. As a simple, yet useful example, consider requiring the designer to only use properly defined stereotypes with classes. In the context of this work, this variant of a stereotype conformance rule from [26] can be implemented using pure OCL (Figure 6), using Python without OCL (Figure 7), and using OCL together with Python (Figure 8).

The stereotypes are defined as in Figure 1. For each class in the user model, its stereotype must be defined accordingly. The example in Figure 6 shows this constraint using OCL extended with the *find* operation. The context for the constraint is a tuple containing the stereotype definition profile in a part called *profile*, and the model to be checked in a part called *model*. The constraint starts by recursively searching for all the classes in the model. For these classes, the following constraint is established: all stereotypes of these classes must be defined in the stereotype definition profile.

The example, albeit longish and not optimized for performance, shows a convenient way of establishing a constraint. The example also illustrates one nice feature of OCL, the ability to navigate over sets. For example, on line three, navigation *stereotype.name* refers to the names of *all* stereotypes of *all* the fetched classes. This makes it easy to state a constraint for a series of attributes.

To perform the same checking without using OCL, we assume a UML metamodel compliant data model with the existence of navigation operations. We will also assume, that a function *findAllClasses* exists that retrieves all classes under the package hierarchy. While it is nowhere near as powerful as *find* with the matching criteria expressed in OCL, it will suffice for this example.

Figure 7 shows the pure Python implementation for finding the names of the user-defined stereotypes. The function performs a search operation similar to the one defined in Figure 6, lines 4 to 8. In addition, for the sake of simplicity the pure Python implementation assumes that a single stereotype exists for every UML element, and that there are only binary dependencies. While the latter assumption usually holds in practice, the former certainly does not. In this example, however, we can circumvent the problem by defining suitable well-formedness constraints for a stereotype definition profile. It becomes evident that there is a need for more advanced mechanisms for navigating the models. OCL quite naturally satisfies this need. We omit the actual stereotype checking procedure.

Finally, Figure 8 shows the same stereotype conformance check implemented using Python with OCL queries. We assume the same data model as in the previous example. The queries are enabled through the use of two methods, *find* and *select*, accessible through the list type. Both operations take one argument, a string containing the matching criteria as an OCL expression. The variable *element* refers to the iterator of the operation. While considerably more compact and readable than its pure Python counterpart, it breaks up the pure OCL code nicely and extends it with the possibility of user interaction and model manipulation. Please also note that the example retrieves the defined stereotypes *and* performs the validity check on the model, whereas the Python example in Figure 6 does only the first part.

6 Model Processing Platform Implemented

We have developed a software platform for UML model processing, called xUMLi. The platform supports authoring of model processing operations by providing an environment with a UML metamodel compliant data model. Our

```
1.  def findAllowedStereotypes( profile ):
2.      allowedStereotypes = []
3.      candidates = findAllClasses( profile )
4.      for cls in candidates:
5.          for dep in cls.clientDependency:
6.              if len(dep.stereotype) > 0:
7.                  # assume single stereotype
8.                  if dep.stereotype[0].name=="stereotype":
9.                      # Here, assume that dependencies are binary
10.                     bcls = dep.supplier[0]
11.                     if bcls.Class=="Class" and \
12.                     len(bcls.stereotype)>0 and \
13.                     bcls.stereotype[0].name=="metaclass":
14.                         # assume single stereotypes
15.                         allowedStereotypes.append( cls.name )
16.     return allowedStereotypes
```

Fig. 7. Function for fetching user-defined stereotypes without using OCL

```
1.  st = profile.find( \
2.      "element.oclAsType(Class).clientDependency->exists(cd |" \
3.      "cd.stereotype.name->includes('stereotype')" \
4.      "and cd.supplier.stereotype.name->includes('metaclass') )" )
5.
6.  for cls in model.find("element.oclIsKindOf('Class')"):
7.      for cst in cls.stereotype:
8.          if len(st.select("element.name = '"+ cst.name+"'"))==0:
9.              # handle class with wrong stereotype
```

Fig. 8. Stereotype conformance checking using embedded OCL with Python

approach is to enable authoring small model operations and combining them together, as if they were the primitive expressions of a very high level model processing language. The chaining of operations can be done with traditional programming languages, or by using a special visual language, VISIOME [18].

VISIOME provides a very high level, visual programming paradigm that relies heavily on the usage of OCL together with a set of fundamental programming constructs, and is especially useful when defining software process related model processing functionality. The environment is not dependent of any specific CASE-tool, but offers a plug-in interface for components that

```
1.    st = profile.find( \
2.          "element.clientDependency->exists(cd |" \
3.          "cd.stereotype.name->includes('stereotype')" \
4.          "and cd.supplier.stereotype.name->includes('metaclass') )" )
5.
6.    for cls in model.find("element.metaclass->includes('Class')").ToList():
7.          for cst in cls.Get("stereotype"):
8.                if st.select("element.name->includes("" +cst.name +"") \
8b.                     .Length==0:
9.                     # handle class with wrong stereotype
```

Fig. 9. Stereotype conformance checking using Python enhanced with OCL

transfer models between a tool repository and the data model. It is therefore possible to support several different CASE-tools or UML model repositories. We have built such import/export plug-ins for Rational Rose, XMI, and some proprietary file formats.

In addition to the data model and the visual language, xUMLi contains an OCL interpreter. We chose to offer the interpreter services through an application programming interface instead of embedding OCL directly into a language. OCL expression is passed to the interpreter as a string, and the context as an object of the data model. The interpreter, as well as the classes in the data model, are implemented as Common Object Model (COM) automation classes. Instances of these classes can be accessed from programming languages that support COM as if the instances were native objects of the language. Typically, the individual model operations are constructed using Python, and then combined together with VISIOME to form scripts of more complex functionality. We have also used C++ and Java to implement model operations. Other languages with COM support include Perl and Visual Basic.

Figure 9 shows a real executable example of using Python with our OCL interpreter. The combination of OCL and Python forms a very expressive power user scripting mechanism for performing model processing operations. The example is the same as that in Figure 8, with four minor modifications to fit the specifics of xUMLi. The four differences are explained later in this section. They are marked with **bold** font in the figure.

The OCL interpreter was developed in 2001, and therefore follows the OCL 1.4 specification. The interpreter was intended to be used internally by the VISIOME engine, but was later found very useful for individual model operations as well. Due to the close relationship with the visual language and its simple type structure, the interpreter considers all user defined objects to

be dynamically typed. That is, they can have properties with any name, and each property can contain any number of values. The interpreter is thus unable to perform parse-time checks on the validity of navigations, and will just as happily accept *someClassifier.name* as its mistyped form *someClassifier.nmae*. So, although the xUMLi data model itself is aware of the structure of the UML metamodel, the OCL interpreter is not.

Because of the dynamic typing on objects, the interpreter does not support the *oclIsKindOf* operation. For this reason, a new property, *metaclass*, is included in each object in the xUMLi data model. The property contains metadata about the object, namely the names of the UML metaclasses the object's class is derived from. For example, the *metaclass* in an object of type *Classifier* contains the strings *'ModelElement'*, *'Namespace'*, *'GeneralizableElement'*, and *'Classifier'*. Although this information is metadata, from the point of view of the OCL interpreter it is just an ordinary property. Line 6 in the Figure 9 contains an example where this construct is used instead of *oclIsKindOf*.

As another, perhaps not as obvious, consequence of the application model blindness exhibited, a navigation always results in a set. For example, on line eight in Figure 9 *element.name* results in a set of either one or zero strings, depending on whether the classifier referenced by the iterator *element* has a name or not. Similarly, in order to compare the name of the classifier, we must write *element.name->includes('Class A')*, or *element.name->first() = 'Class A'*, or something similar instead of the much simpler *element.name = 'Class A'*. This is certainly annoying, and it does make the OCL expressions slightly more difficult to read, but it does not affect the expressiveness of the language.

The remaining two differences between Figures 8 and 9 concern collections. On line 6, the *ToList()* method changes an OCL interpreter friendly collection into a list type better suited for Python and other dynamic languages. The read-only property *Length* on line 8b contains the number of items in the collection.

As a deviation from the OCL 1.4 specification, collections can be placed inside other collections, despite the specification's explicit orders to flatten such nested collections. It was clear from quite early on that a set of sets is a very useful structure. Also, prior to the introduction of the tuple abstract data type in OCL 2.0, collection was the only way to present anything resembling a tuple. Banning the use of a set of tuples or a set of sets seemed unnecessary and even harmful, and so the somewhat exotic limit was ignored, along with the automatic flattening of collections.

The OCL interpreter allows extending OCL with new operations. These

extensions are implemented as plug-ins. Operations are placed inside a namespace, and can then be used in OCL expressions as defined in the language specification. We have introduced two new operations, *find* and *flatten*, for collection types. The former is explained in detail in Section 3, and the latter creates one flat collection from a nested collection. This will become obsolete as an extension with the adoption of OCL 2.0, since the same functionality is included in there in an operation with the same name. The extended operations are placed inside a namespace called *std_ext*. For example, the *find* expression on line 6 in Figure 9 would be written in OCL as *model.std_ext::find(e|e.metaclass->includes('Class'))*.

7 Observations on OCL

This section contains observations we have made during our two years of using OCL. We discuss some short comings of OCL as well as areas where it could be improved. The thoughts expressed describe specific problems, but we do not have detailed solutions to propose.

One of the key points for using OCL is ease of use and readability. The OCL specification states that it "is a formal language that remains easy to read and write" [17, pp. 6-2][3, pp. 2-1]. From our experience this is closer to defining a goal than stating a fact. Whenever a new person got in contact with OCL for the first time, they became very confused. Short, straight-forward expressions are often very clear, but when the complexity (and size) grows, the clarity and thus readability decreases radically.

There are probably several factors involved. For example, the users are not as versed in the use of OCL as they are in their preferred programming language. The lack of guidelines for formatting expressions, e.g. indentation rules, might play a role, too, but it is not completely a matter of inexperienced users and badly written OCL. Although the clarity can be improved with comments and variable definitions, it seems that the syntax is prone to hiding the structure of the expressions. Undoubtedly the obscurity of the OCL 1.4 specification is partly responsible for the initial confusion. The readability and the organization of the specification have been greatly improved in the OCL 2.0 draft.

It could be argued, that the types and operations defined in the OCL Standard Library are not very extensive. On the other hand, the extension mechanisms are excellent and offer flexible means to overcome the limitations of the predefined operations. Still, we believe there is a need for better and smarter ways than *allInstances* to locate data in the model, and that such tools should be included in the standard OCL. For example, our extension,

```
parent : Set(GeneralizableElement);
parent = self.generalization.parent

allParents : Set(GeneralizableElement);
allparents = self.parent->union(self.parent.allParents)

[3] Circular inheritance not allowed
not self.allParents->includes(self)
```

Fig. 10. An example of a transitive closure along a navigation

the *find* operation is definitely not very clever, but still it has become the most common way for us to use OCL.

An example of a different kind of an advanced navigation is the transitive closure according to a specific navigation. This is quite a common expression, even the UML Well-Formedness rules are littered with them. For example, rule three for *GeneralizableElement* [17, pp. 2-59], which forbids circular inheritance, uses a transitive closure of the navigation *generalization.parent* [17, pp. 2-60]. The rule and its two helper definitions are shown in Figure 10. We do not propose that these two examples be added to the Standard Library, we present them merely to demonstrate that there is need for more flexible model traversing in the language itself.

Creating operations such as *allParents* for a specific navigation is obviously possible using OCL. Crafting a more general version, one that could traverse along any navigation from any class in the application model, is only possible if we fix the application model. The same is true for our *find* operation. Since OCL lacks any means to access the metadata of the types themselves it is impossible to express such navigations, or to specify advanced navigation techniques that are not tied to the application model.

If the language contained better support for accessing this type data, OCL could be extended using OCL itself even in the case of advanced generic operations. Ability to define extensions in that way would mean that the extensions worked in any OCL interpreter, regardless of its origin. This would further improve portability of OCL expressions themselves. Although portability of extensions is not important at the moment, it might become an issue in the near future, if CASE tools (and software designers) adopt OCL in a larger scale.

8 Related Work

Even before the eve of OCL 2.0 there have been papers where the writers have found, somewhat similar to us, OCL useful for querying information. For instance, Hobart and Malloy discuss using OCL queries for debugging C++ [9], and Marder et al. propose a UML repository and an API, based on UML metamodel, for managing and querying UML models [13]. In the latter environment, OCL constraints can be used to specify and check UML models. The constraint can, for instance, hold design guidelines or semantic invariants to enforce validity of UML models.

There are also a number of other OCL tools meant for model checking. LCI OCL Evaluator (OCLE) [15] is an independent model checking tool meant to be used with any CASE-tool. OCLE uses the XMI format to pass UML models between OCLE and the CASE-tool. USE system [22], developed at the University of Bremen, is a standalone system for the specification of information systems supporting OCL to specify additional integrity constraints on the model. Dresden OCL toolkit [10] consists of several modules that parse, type check, and normalise OCL constraints. It is used, for instance, with Novosoft UML (NSUML) [14] Java library, which is used in various CASE-tools, e.g. ArgoUML [2]. ArgoUML therefore offers OCL support as well, allowing additions of OCL constraints, for which syntax and type checks are implemented. However, model checking is only one set of tasks in model processing. In these tools, usage scenarios like model manipulation are not possible with OCL alone, nor with OCL combined with a programming language.

The ideas behind the following three tools are perhaps closest to ours: a model processing toolkit made in Turku Center of Computer Science [19], the UML all purpose transformer (UMLAUT) [28], and the OCL4Java library of the Kent Modelling Framework (KMF) [12]. In [19], Porres discusses the requirements and most important features of a stand-alone UML-aware programming environment, relying on a Python-like scripting language that can be used to manipulate and extract information from any UML model. The scripting language and the supporting environment are used for loading, querying, modifying, and saving UML models. Interoperability with existing tools is enabled by supporting XMI serialization. In contrast to our approach, OCL is not used as such in their system, but the scripting language is specified to include the functionality of OCL. In their approach, e.g. the well-formedness rules of UML must currently be translated from OCL to their language manually. Further, while they use a specific scripting language to write the operations, our platform allows the usage of independent operations that are not tied with a specific programming language.

Ho et. al. [28] have developed UMLAUT, a framework dedicated to ma-

nipulation of UML models. They use the UML action language to specify transformations, and OCL to express the selection criteria of the transformations. They mix an action language and OCL together to better describe both queries to the models and the manipulation of the models. Although the approach of UMLAUT is close to ours, there are also differences. They use a domain specific language, and concentrate on model transformations. We have also a domain specific language for combining the operations, but the operations can be specified with any language supporting COM, and our tool allows developing practically any kind of program, including arbitrary user interaction and external repository use.

KMF is a set of tools for model driven software development. It can be used, e.g. for defining and checking constraints on visual languages presented as metamodels. OCL4Java, a Java library for parsing and executing OCL expressions, is part of the framework. The library parses a string containing the OCL expression and evaluates it on the context, passed as a Java object. The idea, to provide OCL capabilities through an interface rather than embedding, is the same as ours. The differences between OCL4Java and our interpreter are mainly technical. OCL4Java is a Java library, and since Java as a language is operating system and hardware independent, so is their library. As a trade-off the library can not be used from other languages. Our interpreter can be used from several different languages, but is, unlike OCL4Java, in practise tied to Windows operating systems due to the use of COM.

9 Discussion

In this paper we discussed the needs and possibilities for using OCL for model processing purposes, and stressed the need of domain specific languages in model processing. We especially highlighted the preferability of mixed use of OCL and a programming language in this domain to using OCL or a general purpose programming language alone. According to our experiences, it seems that in practice the most convenient way of using OCL is together with a supporting programming language. This enables usage scenarios outside the scope of OCL (e.g. user interaction, model manipulation), but simultaneously allows the user to use OCL's mechanisms, e.g. for querying, and expressing constraints on, UML models.

We also introduced our CASE-tool independent model processing tool where we have used OCL as an integral part of the model manipulation facilities, and showed how we have applied and extended OCL in our environment to fit for model processing purposes. Our environment supports UML processing components implemented with any programming language supporting

COM. Thus the modeler is able to use the programming languages of her choice.

We have used our environment for couple of years now, and it has proven its suitability for non-trivial model processing tasks and process support. According to our practical experience, using clearly separated OCL expressions for queries, etc. makes the code better organized and easier to comprehend. This improved readability also makes the code easier to maintain, which is very helpful during the early phases of the development.

In addition to using the platform to develop tool support for different processes and performing some case studies, we are also going to further develop the platform in the future. For instance, we will adopt UML 2.0 including OCL 2.0, and develop the OCL support of VISIOME further.

References

[1] Airaksinen, J., K. Koskimies, J. Koskinen, J. Peltonen, P. Selonen and T. Systä, *xUMLi: Torwards a Tool-independent UML Processing Platform*, in: K. Østerbye, editor, *Proceedings of the Nordic Workshop on Software Development Tools and Techniques, 10th NWPER Workshop*, Copenhagen, Denmark (2002), pp. 1–15.

[2] ArgoUML Project, *Argouml project home page* (2003), on-line at http://argouml.tigris.org/.

[3] Boldsoft, R. S. Corporation, IONA and A. Ltd., *Response to the uml 2.0 ocl rfp, version 1.6* (2003), on-line at http://www.omg.org/cgi-bin/doc?ad/03-01-07.

[4] Clarke, S., *Extending standard uml with model composition semantics*, Science of Computer Programming **1** (2002), pp. 71–100.

[5] D. D'Souza and A. Wills, "Objects, Components, and Frameworks with UML," Addison-Wesley, 1999.

[6] Engels, G., R. Heckel, G. Taentzer and H. Ehrig, *A combined reference model- and view-based approach to system specification*, International Journal of Engineering and Knowledge Engineering **4** (1997), pp. 457–477.

[7] Fowler, M., "Refactoring: Improving the Design of Existing Code," Addison-Wesley, 1999.

[8] Henderson-Sellers, B. and B. Unhelkar, "OPEN Modeling with UML," Addison-Wesley, 2000.

[9] Hobatr, C. and B. A. Malloy, *The design of an OCL query-based debugger for C++*, in: *Proceedings of ACM Symposium on Applied Computing*, Las Vegas, USA, 2001, pp. 658–662.

[10] Hussmann, H., B. Demuth and F. Finger, *Modular architecture for a toolset supporting OCL*, in: A. Evans, S. Kent and B. Selic, editors, *UML 2000 - The Unified Modeling Language. Advancing the Standard. Third International Conference*, York, UK (2000), pp. 278–293.

[11] K. Normark, *Synthesis of Program Outlines from Scenarios in DYNAMO* (1998), on-line at http://www.cs.auc.dk/~normark/dynamo.html.

[12] KMF, *Kent modelling framework home page* (2003), on-line at http://www.cs.kent.ac.uk/projects/kmf/index.html.

[13] Marder, U., N. Ritter and H.-P. Steiert, *A DBMS-based approach for automatic checking of OCL constraints*, in: *Proceedings of OOPSLA'99 workshop Rigorous Modeling and Analysis with the UML: Challenges and Limitations*, Denver, Colorado, USA, 1999.

[14] Novosoft, *Novosoft metadata framework and uml library* (2003), on-line at http://nsuml.sourceforge.net.

[15] OCLE, *Lci ocl evaluator project home page* (2003), on-line at http://lci.cs.ubbcluj.ro/ocle/index.htm.

[16] OMG, *Mda guide v1.0* (2003), on-line at http://www.omg.org/docs/omg/03-05-01.pdf.

[17] OMG, *Omg unified modeling language specification, version 1.5* (2003), on-line at http://www.omg.org/uml.

[18] Peltonen, J., *Visual Scripting for UML-Based Tools*, in: *Proceedings of ICSSEA 2000*, Paris, France, 2000.

[19] Porres, I., *A Toolkit for Manipulating UML Models*, in: *TUCS Technical Report No. 441*, Turku, Finland (2002).

[20] Rácz, F. D. and K. Koskimies, *Tool-Supported Compression of UML Class Diagrams*, in: *Proceedings of UML'99*, Fort Collins, Colorado, USA, 1999, pp. 172–187.

[21] Rational Software Corporation, *Rose Enterprise Edition* (2003), on-line at http://www.rational.com/products/rose.

[22] Richters, M. and M. Gogolla, *Validating UML Models and OCL Constraints*, in: *Proceedings of the 3rd International Conference on Unified Modeling Language (UML'2000)* (2000), pp. 265–277.

[23] S. Schönberger and R. K. Keller and I. Khriss, *Algorithmic support for model transformation in object-oriented software development*, Concurrency and Computation: Practise and Experience **5** (2001), pp. 351–383.

[24] Selonen, P., *Set Operations for Unified Modeling Language*, in: *Proceedings of the Eight Symposium on Programming Languages and Tools, SPLST'2003*, Kuopio, Finland (2003), pp. 70–81.

[25] Selonen, P., K. Koskimies and M. Sakkinen, *Transformations between uml diagrams*, Journal of Database Management **3** (2003), pp. 37–55.

[26] Selonen, P. and J. Xu, *Validating uml models against architectural profiles*, in: *Proceedings of the 9th European software engineering conference held jointly with 10th ACM SIGSOFT international symposium on Foundations of software engineering* (2003), pp. 58–67.

[27] Tarja Systä, *Incremental construction of dynamic models for object-oriented software systems*, Journal of Object-Oriented Programming **5** (2000), pp. 18–27.

[28] Wai-Ming Ho, F. Pennaneac'h, N. Plouzeau, *UMLAUT: a framework for weaving UML-based aspect-oriented designs*, in: *In Proc. of 33rd International Conference on Technology of Object-Oriented Languages*, 2000, pp. 324–334.

[29] Whittle, J. and J. Schumann, *Generating Statechart Designs from Scenarios*, in: *Proceedings of the International Conference on Software Engineering, ICSE'00*, Limerick, Ireland, 2000, pp. 314–323.