

On the Need of User-defined Libraries in OCL

Thomas Baar

akquinet tech @ spree GmbH
Bülowstraße 66, D-10783 Berlin, Germany
thomas.baar@akquinet.de

Abstract. Reuse is a fundamental concept of efficient software development. Object-oriented *implementation languages* offer reuse on different levels of granularity: method, class, library. User-defined libraries are widely used to share implementation code among different projects. In contrast to this, the *specification language* OCL merely offers the OCL Standard Library for reuse in different projects. There is no standardized way to import user-defined OCL constraints into another project. In this paper, we argue on the need of a standardized mechanism to make reuse of OCL specifications within a different context possible.

1 Motivation

One indicator for the success of an implementation language is the number of libraries made available. A high number of available libraries is a sign of an active community. Many members of the community are willing to share the achievements they made.

A library should address one recurring programming problem, such as OR-mapping, or logging. By using the library, a programmer can build on top of abstractions provided by the library, what can help to cut down development costs and to improve the quality of the developed system.

Uncontrolled publication of libraries, however, can confuse the community; a well-known example from Java are different logging frameworks such as log4j or Sun's logging API. Fortunately, some powerful mechanisms can avoid the proliferation of different libraries for the same purpose.

In the Java world, many of the widely adopted libraries are authored by organizations whose name became a synonym for high-quality software, e.g. Apache Software Foundation (ASF), Eclipse Foundation, Mozilla, JBoss. Important libraries can become an official standard¹ or part of important library bundles, such as Java EE (Enterprise Edition). These mechanisms as well as training and certification programs ensure a widespread dissemination and usage of important Java libraries today.

For the OCL community, the problem of library proliferation does not exist yet, because there is no standardized way to create or to import a library. Consequently, there is also no market of OCL libraries.

¹ The standardization process is defined in the Java Community Process (JCP).

The contribution of this paper is to show the usefulness of user-defined OCL libraries. We illustrate our arguments using a small example, which is discussed from the perspective of both OCL and Java.

Related Work: The problem of missing library support in OCL has been recently recognized and addressed by Chimiak-Opoka in [1]. Her tool implements an import-statement for OCL, what already enables reuse of OCL constraints. The proposed OCL-import, however, is not aligned yet with the different kinds of UML-import.

2 Running Example

Suppose, you develop an analysis tool for Java code. The ultimate goal of the tool is to find dead code. More precisely, the tool should detect classes, whose methods are never invoked if the system is started via the main-method of the *start class*.

The tool works in two phases: In the *first phase*, information about the control flow is extracted from the Java source code and a *call graph* is built. The call graph shows method invocations as (directed) call dependencies between the calling class and the called class. The underlying data structure of the call graph is shown in Fig. 1. We assume for the rest of this paper that the first phase has been already implemented correctly, i.e. that the call graph is already available.



Fig. 1. Data structure of call graph

In the *second phase*, the tool is supposed to provide two kinds of dead code analysis. The first kind of analysis detects all classes that are never invoked by other classes. In the call graph, such classes are *isolated* in the sense that they do not have any incoming call dependency.

The second kind of analysis detects *orphan classes*. Orphan classes are classes that can have incoming call dependencies from other orphan classes, but no path of call dependencies exists from the start class to an orphan class. Since orphan classes are not reachable from the start class, their code is never executed.

3 Current Solutions in Java and OCL

How can the expected tool behavior be specified/implemented? The tool's functionality is adequately represented by two query methods on class `JavaClass`: `isIsolated():Boolean` and `isOrphan(JavaClass startClass):Boolean`. In the sequel, we will try to implement these two methods in Java and to specify them using OCL.

3.1 Implementation in Java

The implementation of *isIsolated()* is very simple provided that the associations between `JavaClass` and `CallDep` are implemented for each direction by a reference. In this case, the implementation looks as follows:

```
public boolean isIsolated(){
    return incoming.isEmpty();
}
```

The implementation of *isOrphan(JavaClass startClass)* should return `true` if `this` is not reachable from the `startClass` via call dependencies and `false`, otherwise. This requires to compute the transitive closure of the call dependency relationship.

Fortunately, graph problems have been tackled by numerous Java libraries. For example, the open-source library *JGraph*² provides a class `mxGraphAnalysis`, whose methods implement some frequently needed algorithms on directed graphs. The computation of the transitive closure is basically done in `mxGraphAnalysis.getConnectionComponents()`.

If one wants to base the implementation of *isOrphan()* on the library *JGraph*, one should - in a first step - implement a mapping of the instances of `JavaClass/CallDep` to the nodes/edges of a graph in *JGraph*. Then - in a second step - *isOrphan()* can delegate the computation of the transitive closure to `mxGraphAnalysis.getConnectionComponents()`.

3.2 Specification in OCL

The specification of method *isIsolated()* is again very simple:

```
context JavaClass::isIsolated():Boolean
post: result = incoming->isEmpty();
```

For the specification of method `isOrphan()` we would like to reuse a library similar to *JGraph*. Unfortunately, OCL does not support user-defined libraries, yet. Interestingly, there has been attempts to add a transitive closure operator to OCL³, but, at times of writing this paper, this operator has not yet become a part of OCL. Note that if OCL had support for libraries and if a counterpart of the *JGraph* library would be available for OCL, then there would much less need for a separate transitive closure operator in OCL.

4 A Proposal for OCL-Libraries

4.1 Overview

Fig. 2 shows a proposal, how the problem of specifying *isOrphan()* could be solved in an elegant way. The core idea is to have libraries both for OCL specifications and for underlying UML models⁴. Below, we also discuss the obstacles for realizing such a proposal.

² Available from www.jgraph.com

³ See request <http://www.omg.org/issues/issue13944.txt>

⁴ OCL constraints can refer also to non-UML models, e.g. to DSL models. The problems discussed here for UML/OCL libraries apply in a similar fashion also to non-UML/OCL libraries.

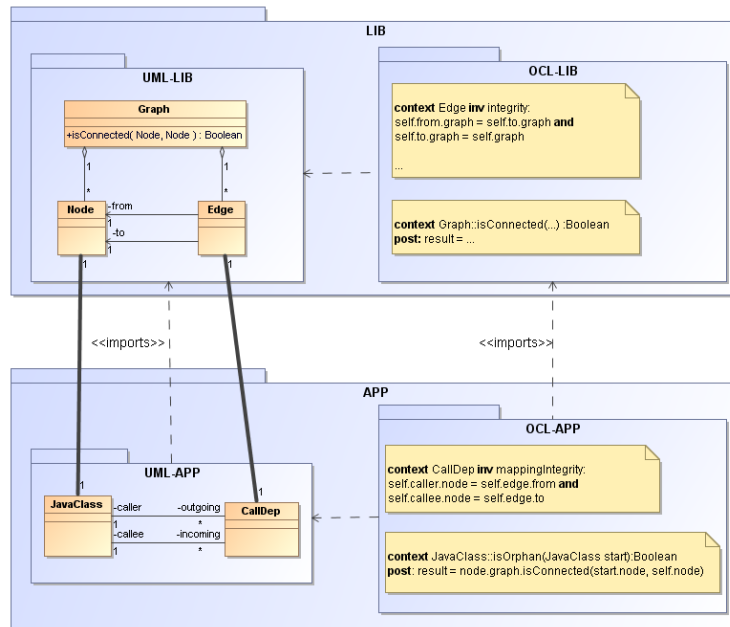


Fig. 2. Proposal for Usage of OCL-Libraries

The upper part of Fig. 2 shows the *library layer* LIB. This layer comprises both OCL constraints (right part) and the underlying UML model (left part). For our running example, an appropriate library would define concepts like `Graph`, `Node`, `Edge` in order to represent graphs. OCL constraints (right part) can fix the intended meaning of the defined concepts, e.g. integrity constraints, and they can specify operations such as `Graph::isConnected()` for the purpose of reuse. Note that the layer LIB itself is nothing but an ordinary UML/OCL model.

The shortcomings of OCL become obvious, if one tries to reuse layer LIB. The bottom part of Fig. 2 shows the reuse of LIB within an *application layer* APP. The UML model UML-APP imports UML-LIB by using *UML's package import*. In our example, the concepts of the application domain (`JavaClass`, `CallDep`) must be mapped to the imported concepts, what is achieved by additional *mapping associations* with multiplicity 1-1. Furthermore, the constraints of OCL-LIB have to be imported. Finally, the operation `isOrphan()` can be specified by a constraint that simply delegates to `Graph::isConnected()` as shown in OCL-APP.

4.2 Obstacles for realizing this proposal

Different UML-imports UML defines different kinds of package import: *import*, *access*, *merge*. Each kind has its own semantics with direct consequences on the usage of imported elements within OCL expressions. Furthermore, it must be possible to *change the library*

after its import. In our example, the library concepts **Node**, **Edge** became endpoints of the mapping associations.

OCL-import OCL does not know an import-statement, yet. The semantics of such OCL-import must correspond with the import of the underlying UML models.

Customization of imports for UML, OCL Experience in (object-oriented) programming shows that libraries can be reused much more flexible if the imported entities can be redefined and adapted towards the needs of the importing context. For UML/OCL-imports, several kinds of customization are imaginable, e.g. merge of model elements and (de)selection of constraints.

5 Conclusions

Specification languages such as OCL are supposed to work on a higher level of abstraction compared with implementation languages. A higher abstraction level means less details to deal with and to use much simpler data structures.

At the level of implementation languages, sharing useful *abstractions* among projects is done by publishing a library. Libraries on important topics have a managed publication lifecycle, i.e. they are specified and reviewed prior to publication.

OCL knows only one library named OCL Standard Library. Sometimes, this library does not offer the data structures one would wish. For the specification of `isOrphan()` one could argue, that this specification could be done easily, as soon as the transitive closure operator becomes part of OCL. This is correct, but the point is that Java does not offer an operator for transitive closure either. As our example shows, implementing `isOrphan()` in Java does not cause any problem provided that an appropriate library is used.

A serious problem of OCL is its inability to make useful *abstractions* available to other projects. Today, the only possibility to share new abstractions among different projects is to let them become a part of OCL Standard Library. Still, adding a new element to OCL Standard Library is comparable to adding a new element to the `java.util` package in Java, what is only in very rare cases an appropriate solution.

What could obviously help are user-defined libraries. As a first step, the OCL language definition has to be extended by mechanisms for defining and importing OCL libraries. Note that these mechanisms have to be supported by each OCL tool. This could pave the way for a - so far missing - market of OCL libraries. Once a vivid market of reusable OCL libraries has emerged, the OCL community could agree in a second step on mechanisms to avoid proliferation of libraries.

References

1. Joanna Chimiak-Opoka. OCLLib, OCLUnit, OCLDoc: Pragmatic extensions for the object constraint language. In Andy Schürr and Bran Selic, editors, *MoDELS*, volume 5795 of *Lecture Notes in Computer Science*, pages 665–669. Springer, 2009.