Electronic Communications of the EASST
Volume 36 (2010)

Proceedings of the
Workshop on OCL and Textual Modelling
(OCL 2010)

On the Need of User-defined Libraries in OCL

Thomas Baar

10 pages

Guest Editors: Jordi Cabot, Tony Clark, Manuel Clavel, Martin Gogolla
Managing Editors: Tiziana Margaria, Julia Padberg, Gabriele Taentzer
ECEASST Home Page: http://www.easst.org/eceasst/ ISSN 1863-2122

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. While encapsulation of implementation code within methods and classes enables reuse within a project, user-defined libraries are widely used to share implementation code among different projects.

The *specification language* OCL offers language concepts like *defined attributes* and *defined operations* to enable reuse within a project. However, reuse among different projects is not possible since OCL does not support the concept of user-defined libraries. 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.

Keywords: software reuse, library management, OO programming, OCL, Java

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 of an implementation language usually addresses one recurring programming problem, such as *OR-mapping*, *graph rendering*, 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 standard[1] 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.

The current version of the Object Constraint Language (OCL 2.2 [OMG10a]) does not support the concept of user-defined libraries. This has both positive and negative consequences. On the

[1] The stardardization process is defined by the Java Community Process (JCP), see [Ora10] for details.

positive side, one could argue that there is no need for the OCL community to think about a process to control the lifecycle of libraries, i.e. there is no need to invent something like the *OCL Community Process*. Furthermore, OCL users are prevented from the pain of having to choose the most suitable library from a set of competing libraries targeting the very same problem. On the negative side, there is no standardized means to share OCL constraints between different projects. Consequently, there is no market of user-defined OCL libraries where different authors try to convince the OCL community that their library is the most elegant solution for an identified problem.

The contribution of this paper is to stress the point that OCL's current mechanisms for reuse are not sufficient. We argue on the need of user-defined libraries and propose an architecture for OCL libraries. A support of user-defined libraries would result in a much more flexible usage of OCL, since the users are not bound to the limited set of data structures and operations offered by the OCL Standardized Library. We illustrate our arguments using a small example, which is discussed from the perspective of both OCL and Java.

The paper is organized as follows. In Section 2 we formulate requirements for a system to be built. This system will serve as a running example for the rest of the paper. In Section 3 the informally described functionality of the running example is both specified using OCL and implemented in Java. We compare the effort for writing both OCL specification and Java implementation. We draw already the conclusion that writing the OCL specification would be less painful, if OCL had a support for user-defined libraries. Section 4 presents a possible architecture for library support in OCL and discusses its merits and limitations. While Section 5 reports on alternative approaches found in the literature and gives further examples that underpin the need of user-defined libraries in OCL, Section 6 summarizes the paper and draws conclusions.

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 analysis tool works in two phases.

In the *preparation 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 call graphs is modeled by a UML class diagram as shown in Figure 1. A call graph (represented by `CallGraph`) consists of Java classes (`JavaClass`) and call dependencies (`CallDep`). The two associations between `JavaClass` and `CallDep` indicate, which Java classes are actually connected by a call dependency. Finally, an integrity constraint makes the model more comprehensible. This constraint says that a call dependency belongs to the same call graph as the Java classes it connects. The constraint is formulated in OCL in the comment box in Figure 1.

We assume for the rest of the paper that the preparation phase has already been implemented correctly, i.e. that the call graph is available.

In the *analysis phase*, the analysis 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

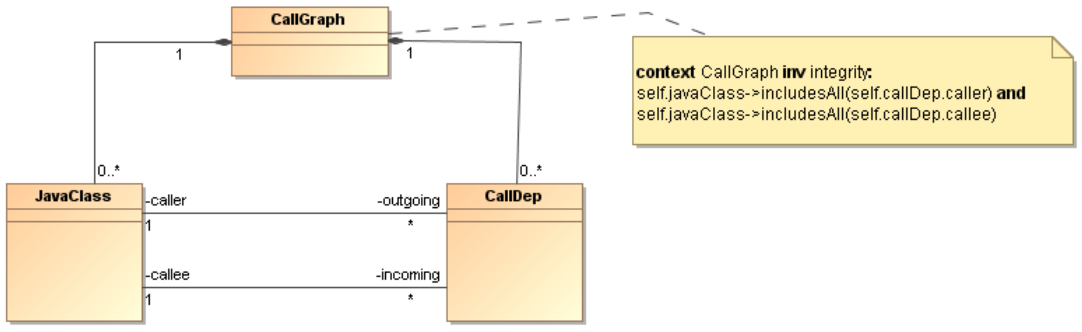
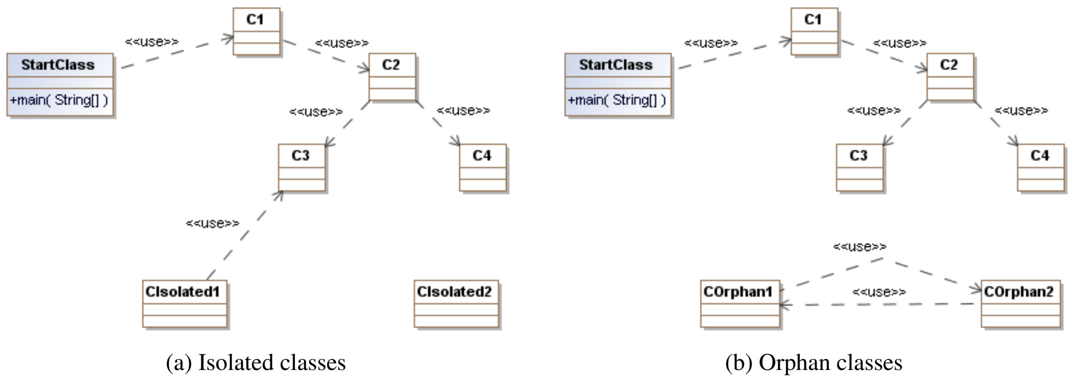

CallGraph
1
1
context CallGraph inv integrity:
self.javaClass->includesAll(self.callDep.caller) and
self.javaClass->includesAll(self.callDep.callee)
0..*
0..*
JavaClass
-caller
1
-callee
1
-outgoing
*
-incoming
*
CallDep

Figure 1: Data structure of call graph

graph, such classes are *isolated* in the sense that they do not have any incoming call dependency. In the example given in Figure 2a, the classes `CIsolated1` and `CIsolated2` are isolated according to this definition.

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. In the example given in Figure 2b, the classes `COrphan1` and `COrphan2` are orphan classes.



(a) Isolated classes (b) Orphan classes

Figure 2: Two kinds of analysis. The start class of the analyzed system is colored in gray

3 Current Solutions in Java and OCL

How can the expected behavior of the analysis tool be specified/implemented? The tool's functionality is adequately represented by the two query methods `isIsolated():Boolean` and `isOrphan(JavaClass startClass):Boolean` on class `JavaClass`. The method call *o.isIsolated()* returns *true*, iff *o* represents an isolated Java class. Analogously, the method *o.isOrphan(start)* returns *true*, iff *o* represents an orphan class wrt. *start*, i.e. there is no path of call dependencies from *start* to *o*.

In the sequel, we will try to realize these two methods both in Java and OCL.

3.1 Implementation in Java

3.1.1 isIsolated()

The implementation of *isIsolated()* is very simple provided that the associations between classes `JavaClass` and `CallDep` are implemented for each direction by a reference. In this case, the implementation looks as in Listing 1.

Listing 1: Java implementation of isIsolated()

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

3.1.2 isOrphan()

The implementation of *isOrphan(JavaClass startClass)* requires to compute the transitive closure of the call dependency relationship.

The computation of the transitive closure is a well-known graph problem. Fortunately, graph problems have been tackled by numerous Java libraries. For example, the open-source library *JGraph* [JGR10] 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()`.

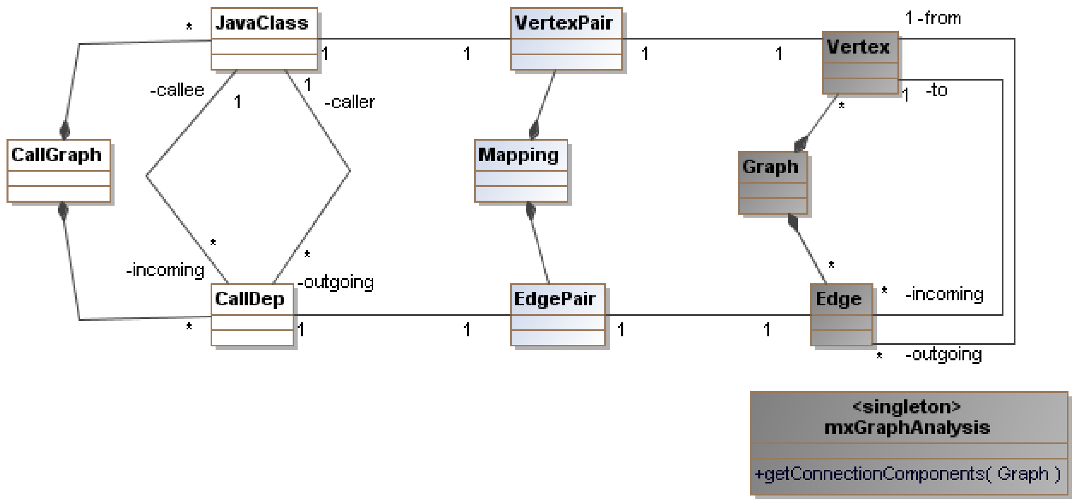


Figure 3: Implementation of running example when using library JGraph

Figure 3 shows the architecture when using the library. On the right hand side of the figure, the classes of the JGraph library are shown (marked by a dark gray background). On the left hand side, one finds the classes of our application (white background), which use the library classes. In the middle, there are some mapping classes (light gray background).

Following the architecture of Figure 3, the main challenge when using JGraph is to keep the data structure of the application (left hand side) with the data structure of the library (right hand side) in sync. Whenever objects of classes `JavaClass` and `CallDep` are created, it must be ensured that corresponding objects of the library classes `Vertex` and `Edge` together with objects of the mapping classes `VertexPair` and `EdgePair` are created as well. The same effort of synchronization is necessary when objects of `JavaClass` and `CallDep` change or when they are deleted.

Once the synchronization of application and library objects is done, the implementation of *isOrphan()* is a simple delegation as shown in Listing 2.

Listing 2: Java implementation of isOrphan()

```
public boolean isOrphan(JavaClass start){
  Graph graph =
          this.getVertexPair().getVertex().getGraph();
  return mxGraphAnalysis.getConnectionComponents(graph).
      differ(this.getVertexPair().getVertex(),
             start.getVertexPair().getVertex());
}
```

3.2 Specification in OCL

3.2.1 isIsolated()

The specification of method *isIsolated()* in OCL is as simple as the corresponding implementation in Java. The specification is shown in Listing 3.

Listing 3: OCL specification of isIsolated()

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

3.2.2 isOrphan()

For the specification of method `isOrphan()` we would like to reuse a library similar to *JGraph*. Unfortunately, OCL does not support user-defined libraries.

Interestingly, there has been attempts to add a transitive closure operator to OCL[2], but, at times of writing this paper, this operator is not a part of OCL. Listing 4 shows the definition of *isOrphan* if OCL had already support for the transitive closure operator. Note that in Listing 4

[2] See request http://www.omg.org/issues/issue13944.txt

we use *TC* as a concrete syntax representation of the transitive closure operator. This, however, is just a 'private notation', because no decision of the OMG, the standardization committee for OCL, has been yet taken on whether and in which form the transitive closure operator will be made available in OCL.

Listing 4: OCL specification of isOrphan() using transitive closure operator TC

```
context JavaClass::isOrphan(start:JavaClass):Boolean
   post: result = not start.TC(outgoing)->includes(self)
```

It should be noted that there would be no need to integrate the transitive closure operator into OCL (and, by doing this, to give up the goal to have a loose semantics for standard OCL, see also [Baa03]), if (1) OCL had support for user-defined libraries and (2) if there would be an OCL library analogously to JGraph available.

4 A Proposal for OCL Libraries

In the last section, we tried to find an ad-hoc solution for the problems illustrated by the running example. As a serious source of problems we identified the missing support for user-defined libraries. In this section, a proposal for an architecture of user-defined OCL libraries is presented.

4.1 Overview

Figure 4 shows a possible architecture of OCL libraries using an example. As example we have chosen the library-based solution for specifying *isOrphan()*. The core idea is to have libraries both for OCL specifications and for underlying UML models[3].

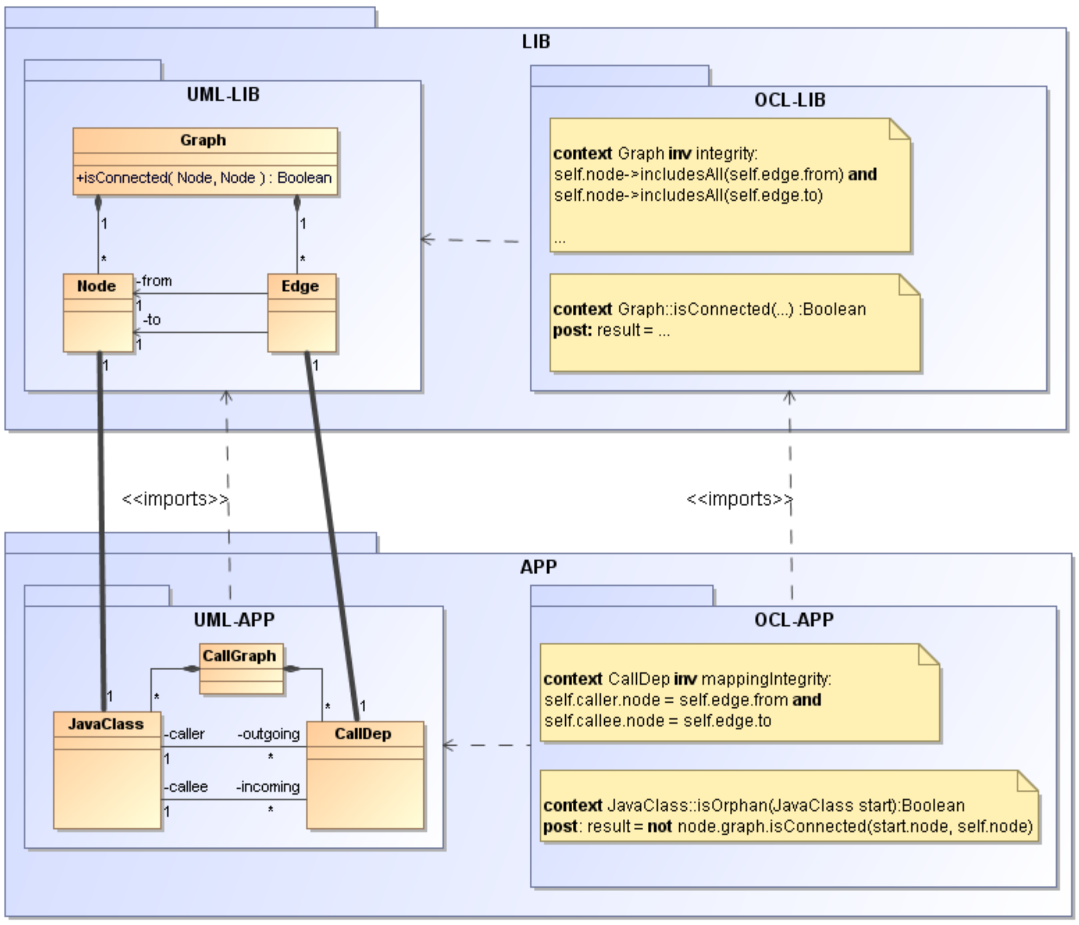
The upper part of Figure 4 shows the *library layer LIB*. This layer comprises both OCL constraints (right part) and the underlying UML model (left part). For our running example, an appropriate library would define concepts like `Graph`, `Node`, `Edge` in order to represent graphs. OCL constraints (right 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 bottom part of Figure 4 shows the reuse of *LIB* within the *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

The following obstacles must be taken into account when realizing this proposal:

[3] OCL constraints can also refer to non-UML models, e.g. to DSL models. The problems for UML/OCL libraries discussed here apply analogously also to non-UML/OCL libraries.



```
context Graph inv integrity:
self.node->includesAll(self.edge.from) and
self.node->includesAll(self.edge.to)
...
```

```
context Graph::isConnected(...) :Boolean
post: result = ...
```

```
context CallDep inv mappingIntegrity:
self.caller.node = self.edge.from and
self.callee.node = self.edge.to
```

```
context JavaClass::isOrphan(JavaClass start):Boolean
post: result = not node.graph.isConnected(start.node, self.node)
```

Figure 4: Proposal for an architecture of OCL libraries

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 support an import-statement. The semantics of such OCL-import must correspond with the different kinds of import of the underlying UML models.

Customization of imports for UML, OCL When reusing libraries in object-oriented programming languages, flexibility is considerably improved by the possibility to redefine and adapt imported entities towards the needs of the importing context. For UML/OCL-imports, comparable kinds of customization are imaginable, e.g. *merge of model elements* and *(de)selection of constraints*.

5 Related Work

The problem of missing library support in OCL has been recently recognized and addressed by Chimiak-Opoka in [CO09]. Her tool *OCL editor* [ST10] implements an import-statement for OCL, which enables reuse of OCL constraints. Currently, Chimiak-Opoka and her team are extending *OCL editor* in a way that it can process the running example presented in this paper.

According to Cabot et al. [CMPT09], statistical functions play an important role in certain domains. They describe in their paper how statistical functions such as *max()*, *avg()*, which compute the maximum and average value of a given OCL collection, could be formally defined in terms of OCL functions. They conclude that these functions should become a part of the OCL Standard Library, but this would be a rather long and laborious process, since the OMG as the standardization authority had to be convinced on the general usefulness of statistical functions. If OCL had already support for user-defined libraries, Cabot et al. could have published an OCL library containing all the functions discussed in the paper. Consequently, everybody could straight away reuse the functions described in [CMPT09] in every OCL tool, propose extensions, give feedback, etc. All this without having to wait for the outcome of a long-winded standardization process.

In [AHM06], Akehurst et al. discuss an interesting approach to generalize the idea of libraries in OCL. They start with the observation that OCL is used as a constraint language not only for UML but for many MOF-based modeling languages. To make these modeling languages amenable to OCL constraints, these languages have to provide certain interfaces. Some of these interfaces, however, are purely related to the OCL Standard Library, e.g. the modeling language has to provide interfaces for primitive OCL types such as *String* and *Real*. Akehurst et al. propose to let a modeling language not only implement interfaces for primitive types, but to let the modeling language completely or partially implement the OCL Standard Library. Analogously, the modeling language could implement any other user-defined library for OCL. Such an OCL library could be reused at least by any other user of the same modeling language, because the library is now part of the modeling language.

Since we use for the Java implementation of the running example the Java library *JGraph*, it will be tempting to name the corresponding OCL library *OCLGraph*. However, one has to be aware that for the scheduling and planning community, *OCLGraph* denotes a generator for planning graphs [SML00]. These planning graphs take object representations for entities of the planning domain into account. These objects are formalized using the *object-centred language* [MP97], which provides concepts such as *sort hierarchy*, *predicates*, *sub-state class definitions*, *invariants*, *operators*. While these concepts play also an important role for the Object Constraint Language, the object-centred language is a different language. To make the confusion perfect, both *Object Constraint Language* and *object-centred language* are abbreviated by *OCL*.

6 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. Successful libraries must have a managed lifecycle, i.e. they are specified and reviewed prior to publication and change.

OCL does not yet support the concept of user-defined libraries. OCL Standard Library is the only available library in OCL. Sometimes, this library does not offer the data structures one would wish. Today, the only possibility to share new abstractions among different projects is to add them to the OCL Standard Library. However, adding a new element to OCL Standard Library is comparable to adding a new element to the `java.util` package. This is, however, not an appropriate solution in most cases.

As discussed in Subsection 3.2, there is currently an open request to the OMG to include the transitive closure operator to OCL. This request would become superficial if OCL had support for user-defined libraries and if a library similar to JGraph would be made available in OCL. Note that the latter variant would be a much more general solution and would alleviate also many other problems indicated by other authors, e.g. [CMPT09].

For these reasons, we propose — in a first step — to extend the OCL language definition 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. One possible action in this direction is the adoption of a library management process similar to the Java Community Process.

Bibliography

[AHM06] D. H. Akehurst, W. G. J. Howells, K. D. McDonald-Maier. UML/OCL Detaching the Standard Library. Pp. 205–212 in [DCGW06].

[Baa03] T. Baar. The Definition of Transitive Closure with OCL – Limitations and Applications. In *Proceedings, Andrei Ershov Fifth International Conference, Perspectives of System Informatics, Novosibirsk, Russia*. LNCS 2890, pp. 358–365. Springer, July 2003.

[CO09] J. Chimiak-Opoka. OCLLib, OCLUnit, OCLDoc: Pragmatic Extensions for the Object Constraint Language. Pp. 665–669 in [SS09].

[CMPT09] J. Cabot, J.-N. Mazon, J. Pardillo, J. Trujillo. Towards the Conceptual Specication of Statistical Functions with OCL. Pp. 7–12 in [YER09].

[DCGW06] B. Demuth, D. Chiorean, M. Gogolla, J. Warmer (eds.). *OCL for (Meta-) Models in Multiple Application Domain, Workshop co-located with MoDELS 2006, Genova, Italy, October, 2006. Proceedings*. Technical Reports TUD-FI06-04. Technische Universität Dresden, Fakultät Informatik, 2006.

[JGR10] JGraph Ltd. JGraph, Version 1.4.0.8. Nov 2010. Available from http://www.jgraph.com.

[MP97] T. L. McCluskey, J. M. Porteous. Engineering and Compiling Planning Domain Models to Promote Validity and Efficiency. *Artif. Intell.* 95(1):1–65, 1997.

[OMG10a] OMG. Object Constraint Language, Version 2.2. Feb 2010. Available from http://www.omg.org/spec/OCL/2.2.

[OMG10b] OMG. OMG Unified Modeling Language (OMG UML), Infrastructure, Version 2.3. May 2010. Available from http://www.omg.org/spec/UML/2.3.

[OMG10c] OMG. OMG Unified Modeling Language (OMG UML), Superstructure, Version 2.3. May 2010. Available from http://www.omg.org/spec/UML/2.3.

[Ora10] Oracle. Java Community Process (JCP). Nov 2010. Available from http://www.jsp.org.

[PUK00] *Proceedings of the 14th Workshop "New Results in Planning, Scheduling and Design" (PuK2000), Berlin, 21-22 August 2000*. 2000.

[SML00] R. M. Simpson, T. L. McCluskey, D. Liu. OCL-Graph: Exploiting Object Structure in a Plan Graph Algorithm. In [PUK00].

[ST10] SQUAM-Team. SQUAM framework / OCL editor. Nov 2010. Available from http://squam.info/.

[SS09] A. Schürr, B. Selic (eds.). *Model Driven Engineering Languages and Systems, 12th International Conference, MODELS 2009, Denver, CO, USA, October 4-9, 2009. Proceedings*. Lecture Notes in Computer Science 5795. Springer, 2009.

[YER09] E. Yu, J. Eder, C. Rolland (eds.). *Proceedings of the Forum at the CAiSE 2009 Conference, Amsterdam, The Netherlands, 8-12 June 2009*. 2009.