ACM/IEEE 17th International Conference on
Model Driven Engineering Languages and Systems

September 28 – October 3, 2014 • Valencia (Spain)

OCL 2014 – 14th International Workshop on OCL and Textual Modeling: Applications and Case Studies

Workshop Proceedings

Achim D. Brucker, Carolina Dania, Geri Georg, and Martin Gogolla (Eds.)

Published on Oct 2014 v1.0

Editors' addresses:

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

Carolina Dania, IMDEA Software Institute, Spain, carolina.dania@imdea.org

Geri Georg, Colorado State University, USA, georg@cs.colostate.edu

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

Organizers

Achim Brucker (co-chair)	SAP SE (Germany)
Carolina Dania (co-chair)	IMDEA Software Institute (Spain)
Geri Georg (co-chair)	Colorado State University (USA)
Martin Gogolla (co-chair)	University of Bremen (Germany)

Program Committee

Michael Altenhofen	SAP SE (Germany)
Thomas Baar	University of Applied Sciences Berlin (Germany)
Mira Balaban	Ben-Gurion University of the Negev (Israel)
Tricia Balfe	Nomos Software (Ireland)
Fabian Buettner	Ecole des Mines de Nantes (France)
Achim D. Brucker	SAP SE (Germany)
Jordi Cabot	INRIA-Ecole des Mines de Nantes (France)
Yoonsik Cheon	University of Texas (USA)
Dan Chiorean	Babes-Bolyai University (Romania)
Robert Clariso	Universitat Oberta de Catalunya (Spain)
Tony Clark	Middlesex University (UK)
Manuel Clavel	IMDEA Software Institute (Madrid, Spain)
Carolina Dania	IMDEA Software Institute (Madrid, Spain)
Birgit Demuth	Technische Universitat Dresden (Germany)
Marina Egea	Atos Research, Madrid (Spain)
Geri Georg	Colorado State University, Fort Collins (Colorado, USA)
Martin Gogolla	University of Bremen (Germany)
Pieter Van Gorp	Eindhoven University of Technology (The Netherlands)
Heinrich Hussmann	LMU Munchen (Germany)
Tihamer Levendovszky	Vanderbilt University (USA)
Shahar Maoz	Tel Aviv University (Israel)
Istvan Rath	Budapest University of Technology and Economics (Hungary)
Bernhard Rumpe	RWTH Aachen (Germany)
Shane Sendall	Snowie Research SA (Switzerland)
Michael Wahler	ABB Switzerland Ltd Corporate Research (Switzerland)
Claas Wilke	Technische Universitat Dresden (Germany)
Edward Willink	Willink Transformations Ltd. (UK)
Burkhart Wolff	Univ Paris-Sud (France)
Steffen Zschaler	King's College, London (UK)

Additional Reviewers

Gábor Bergmann
Gábor Szárnyas
Dimitri Plotnikov
Igal Khitron
Carsten Kolassa

Table of Contents

Preface

Many modeling languages, such as the Unified Modeling Language (UML), advocate the use of graphical notations for modeling. While such visual representations provide a intuitive way of modeling and, usually, allow for describing high-level concepts very nicely, the visual representations are often not suited for describing systems in a precise and unambiguous way: either the visual representations lacks the necessary constructs completely or the visual representations, including all formal details, gets overpopulated.

The challenges of providing both a graphical notation that allows the intuitive modeling of the overall system while still being able to express system properties in a precisely and unambiguously motivated the development of textual specification languages that integrate, extend, or even replace graphical notations. Typical examples of such languages are OCL, textual MOF, Epsilon, and Alloy. Textual modeling languages have their roots in formal language paradigms like logic, programming and databases.

The goal of this workshop was to create a forum where researchers and practitioners interested in building models using OCL or other kinds of textual languages could directly interact, report advances, share results, identify tools for language development, and discuss appropriate standards. The close interaction enabled researchers and practitioners to identify common interests and options for potential cooperation.

Every accepted paper was reviewed by at least three members of the program committee. In addition, these proceedings contain one unreviewed paper summarizing the panel discussion about proposal for future improvements of the OCL. For this paper, each panelist contributed one sections that motivates his or her proposal for extending the OCL.

October 2014 Achim D. Brucker, Carolina Dania, Geri Georg, and Martin Gogolla

Textual, executable, translatable UML[⋆]

Gergely Dévai, Gábor Ferenc Kovács, and Ádám Ancsin

Eötvös Loránd University, Faculty of Informatics, Budapest, Hungary,
{deva,koguaai,anauaai}@inf.elte.hu

Abstract. This paper advocates the application of language embedding for executable UML modeling. In particular, *txtUML* is presented, a Java API and library to build UML models using Java syntax, then run and debug them by reusing the Java runtime environment and existing debuggers. Models can be visualized using the Papyrus editor of the Eclipse Modeling Framework and compiled to implementation languages.
The paper motivates this solution, gives an overview of the language API, visualization and model compilation. Finally, implementation details involving Java threads, reflection and AspectJ are presented.

Keywords: executable modeling, language embedding, UML

1 Introduction

Executable modeling aims at models that are completely independent of the execution platform and implementation language, and can be executed on model-level. This way models can be tested and debugged in early stages of the development process, long before every piece is in place for building and executing the software on the real target platform.

Executable and platform-independent UML modeling changes the landscape of software development tools even more than mainstream modeling: graphical model editors instead of text editors, model compare and merge instead of line based compare and merge, debuggers with graphical animations instead of debuggers for textual programs, etc. Figure 1 depicts the many different use cases such a toolset is responsible for. Models are usually persisted in a format that is hard or impossible to edit directly, therefore any kind of access to the model is dependent on different features of the modeling toolset. Are the available tools ready to meet the requirements? Experience shows that they still need to evolve a lot. Another aspect is that, while graphical notations help understanding software more than textual representations, editing graphics is usually less productive than editing text [13].

Textual modeling languages solve many of these concerns. The many high-quality text editors with sophisticated editing and search-related features, as well as the numerous compare and merge tools serve as a solid basis. However, merely defining a textual notation for modeling does not solve all the issues. Text

---

[⋆] This work is supported by Ericsson Hungary and KMOP-2008-1.1.2.

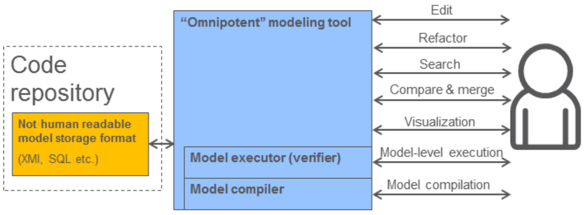


Fig. 1. Using an "omnipotent" modeling framework

editors need plugins to do syntax highlighting and autocompletion correctly for the new language. Executable modeling makes even heavier-weight demands: interpreter and debugger are also required.

Language embedding is a technique to implement a language in terms of an existing one. The implementation language is called the *host language*, while the implemented one is referred to as the *embedded language*. One way to embed a language is to create an API in the host language that offers the constructs of the embedded language for the developers. A program of the embedded language is therefore a host language program that heavily uses this API. The API can, on one hand, implement the execution semantics of the embedded language (making it executable in the runtime environment of the host language) and, on the other hand, can build an internal representation of the embedded program (to be used further for compilation, visualization, analysis etc.). The big advantage is that an embedded language can piggyback on the development and runtime environment of the host language.

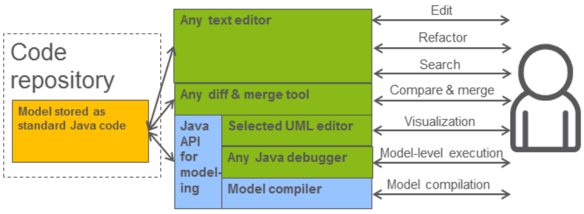


Fig. 2. Using an embedded language for executable modeling

This paper proposes using language embedding for executable UML modeling. As stated above, such an embedded language can be realized by an API that provides the constructs of executable UML. It is natural to use an object-oriented programming language as host language, because many of the UML modeling concepts, such as classes, inheritance, attributes, operations, visibility, instance creation etc. are already present in these languages. Other constructs, like components, associations, state machines, events etc. are usually not explicitly present, therefore these abstractions have to be provided by the API. The implementation of this API has to capture the runtime semantics of these constructs. A UML model can then be implemented by a program in the host language naturally reusing the modeling constructs that are part of the host language and using the API to build the rest of the model. Model execution, in this setup, is not more than running the resulting program in the execution environment of the host language. Figure 2 highlights how many elements of the "modeling environment" are provided for free by this approach. Editing, searching, basic refactoring can be performed in the usual development environment (smart text editor or IDE) of the host language. The debugger of the host language can be reused to observe model execution in detail. Visualization of the model can be achieved via existing UML editors or viewers.

Authors of this paper have created a prototype implementation of a modeling language along these lines [7]. Its name, *txtUML* stands for *textual, executable, translatable UML*. Our host language is Java. Papyrus, the primary UML editor of the Eclipse Modeling Framework, is selected to provide model visualization.

The paper is organized as follows. The next section reviews related work and compares them to our approach. Section 3 presents the language API, while sections 4, 5 and 6 talks about execution, visualization and model compilation. More details on the implementation of these aspects are given in Section 7. The last section summarizes the paper and highlights its most important contributions.

2 Related Work

The UML standard does not define precise execution semantics, but the *Foundational Subset for Executable UML (fUML)* [18] is an OMG standard doing that for a subset of UML, including classes and actions, but excluding components and state machines. There is another OMG standard in preparation to define precise semantics of UML composite structures [20].

BridgePoint [15] is an executable modeling tool originally based on the *Shlaer-Mellor methodology* [24]. Its metamodel is a non-standard fork of UML, including components, classes, state machines (with graphical-only editor) and action language (textual with custom syntax). BridgePoint includes open-sourced [2] editor, and commercial model executor with graphical debugging features and model compiler. A similar, commercial solution is *iUML*, a tool based on the *xUML* methodology [23]. A UML simulator based on a generic model execution engine [14] was developed by IBM Haifa Research Lab and integrated into *Rational Software Architect*. *Topcased* [22] is an open-source joint industry-academia project to create an open-source engineering toolkit. It is implemented over the Eclipse platform and supports UML model execution with graphical feedback. The project is currently under migration to *PolarSys* [8]. The *Eclipse Modeling Framework (EMF)* [25] is surely the most active open source modeling community nowadays. *Moka* [4] is one of the EMF-based tools, and it provides model execution with graphical animations for fUML actions. The *Yakindu Statechart Tools* [10] is an open source toolkit for editing and simulating statecharts and compiling them to C, C++ and Java. It supports statecharts with semantics slightly different from UML state machines and a limited textual action language. All these tools come with graphical-only editors for most layers of their modeling language and are subject to the concerns described in Section 1.

Turning to textual UML modeling, *Action Language for Foundational UML (Alf)* [17] needs to be mentioned. This is an OMG standard that defines textual syntax for fUML. Although there are prototype tools [11] to parse and run Alf code, the support for this language is not mature enough yet. Furthermore, fUML and Alf are based on a limited subset of UML, they miss concepts like components and state machines that are essential for executable modeling. The *Umple* project [12, 9] shows a very neat way to integrate textual modeling with traditional coding. It provides custom syntax for associations, state machines that can be mixed directly into Java, C++, Php or Ruby sources. The Umple compiler preprocesses these files and translates the Umple code fragments to the language they are mixed in. Umple relates to our work because both approaches are textual and try to make modeling more lightweight. On the other hand, their goals are different: Umple provides easy mix-in of model abstractions into traditional programming projects, while txtUML aims at completely platform and implementation language independent, executable modeling.

Section 1 discussed the importance and difficulties of *model management*. The Epsilon [21, 1] project provides a number of textual languages based on a common expression language for different model management tasks like transformation, comparison, merge etc. In particular, not human readable model storage format is one aspect of the model management problem, which is addressed by the OMG standard *UML Human-Usable Textual Notation (HUTN)* [16]. In this paper, in contrast, we explore the possibilities of reusing the management toolset of a mainstream programming language.

Regarding the visualization aspect of our project, *MetaUML* [3] (a LaTeX package to create UML diagrams) and *TextUML* [6] (providing custom textual notation to build UML diagrams) are related projects. Furthermore, there are UML tools (Topcased java2uml importer, ObjectAid, UML-lab, Class Visualizer, MoDisco etc.) that can create UML diagrams (usually class diagrams) based on Java code. The important difference compared to our approach is that these tools start from arbitrary Java code and produce a small part of a model on a best effort basis, while our library only accepts a subset of Java (the txtUML embedded language) and turns the full model definition (including state machines, actions etc.) to a model for visualization or compilation.

3 Language Overview

Our implementation currently supports classes, state machines and action code. To make clear distinction between plain Java classes and those that are part of the UML model description, all model classes have to inherit (directly or indirectly) from `ModelClass`, which is provided by our API[1]. Associations are also represented by class definitions, inheriting from `Association`. Multiplicities are described as Java annotations like `@One`, `@Many` etc. For example, a class diagram talking about *users* using *machines* can be described this way:

```
class Machine extends ModelClass { /* ... */ }
class User extends ModelClass { /* ... */ }
class Usage extends Association {
  @One Machine usedMachine;
  @Many User userOfMachine;
}
```

Signals are defined by classes that inherit from `Signal`. Let us define a signal that users can send to machines by pressing the power button:

```
class ButtonPress extends Signal {}
```

State machines are described by nested classes inside model classes, inheriting from `State`, `InitialState` or `Transition`. The annotations `@From`, `@To` and `@Trigger` complete the definition of transitions.

```
class Off extends State { /* ... */ }
class On extends State  {
  public void entry() { /* ... */ }
  public void exit() { /* ... */ }
}

@From(Off.class) @To(On.class) @Trigger(ButtonPress.class)
class SwitchOn extends Transition {
  public void effect() { /* ... */ }
}
```

Operations of model classes, entry and exit actions of states and effects of transitions contain action code. The `doWork` operation of the `User` model class, for example, log a message to the console, selects the machine that it is associated with, then presses its power button:

```
void doWork() {
  Action.log("User: starting to work...");
  Machine myMachine = Action.selectOne(this, Usage.class, "usedMachine");
  Action.send(myMachine, new ButtonPress());
}
```

Action language constructs include instance related operations (eg. creation, deletion, access to operations and attributes), association related ones (link, unlink, different traversals), signal sending, logging and basic operations of elementary types like integers and strings.

It is important to note that only a restricted subset of Java is allowed in model definitions. For example, adding a `Vector<User>` attribute into the `Machine`

---

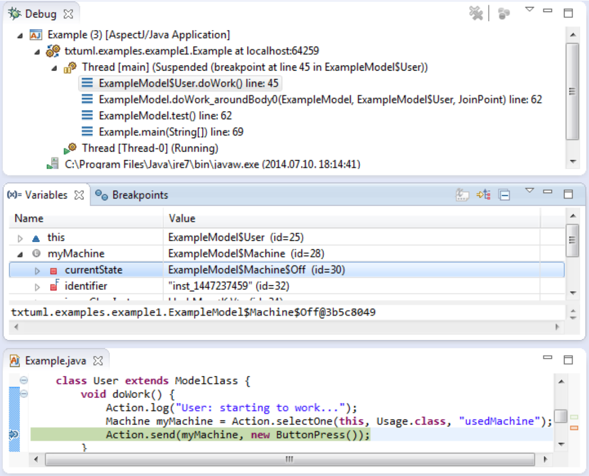
[1] Inheritance between two model classes encodes UML generalization. Java does not support multiple inheritance, which is currently a limitation of txtUML.

class instead of the explicit definition of the association is invalid: Selecting the concrete collection type is an optimization issue and should not be part of the abstract model. The techniques to be discussed in Section 7.2 can rule out the violations of these rules, including constraints specified in the UML standard.

Using an embedded language in Java makes model descriptions a bit more verbose than tailor-made custom syntax. This is a moderate price for familiar syntax and the possibility of model execution and debugging with existing tools.

4 Execution

A model, defined using the API introduced above, can be compiled, run and debugged as any Java application. Figure 3 shows a debugging session executing the example used in Section 3. A breakpoint, set on the signal sending instruction of the `doWork` method, is being hit. It is possible to inspect the threads that belong to the active object instances, the attributes of these instances, in particular the current state of the machine instance. When stepping over the send instruction in the debugger, the developer can verify that the state changes from *off* to *on*.



```
Example (3) [AspectJ/Java Application]
  txtuml.examples.example1.Example at localhost:64259
    Thread [main] (Suspended (breakpoint at line 45 in ExampleModel$User))
      ExampleModel$User.doWork() line: 45
      ExampleModel.doWork_aroundBody0(ExampleModel, ExampleModel$User, JoinPoint) line: 62
      ExampleModel.test() line: 62
      Example.main(String[]) line: 69
    Thread [Thread-0] (Running)
  C:\Program Files\Java\jre7\bin\javaw.exe (2014.07.10. 18:14:41)
```

Name	Value
this	ExampleModel$User (id=25)
myMachine	ExampleModel$Machine (id=28)
currentState	ExampleModel$Machine$Off (id=30)
identifier	"inst_1447237459" (id=32)

```
txtuml.examples.example1.ExampleModel$Machine$Off@3b5c8049
```

```
    class User extends ModelClass {
        void doWork() {
            Action.log("User: starting to work...");
            Machine myMachine = Action.selectOne(this, Usage.class, "usedMachine");
            Action.send(myMachine, new ButtonPress());
        }
```

Fig. 3. Eclipse debugging session executing the example of Section 3

Discussing the execution semantics behind our implementation is far beyond the scope of this paper. We summarize our approach as follows: We start from the constructs in the xtUML modeling language [2], port them to standard UML2 [19] and gradually add more UML2 abstractions. In case of abstractions that are covered by other standards, like fUML [18] and PSCS [20], we adopt the standard semantics. If the execution semantics of the considered abstractions are not standardized (eg. state machines), we closely follow the informal semantics given in the UML2 standard, examine different possibilities and select one based on usability and implementation considerations.

5 Visualization

Even if our proposal is a textual modeling language, we consider visualization an extremely important aspect. Graphical representation of system architecture and behavior makes it easier to understand large systems. The advantage of our approach is that we have the possibility to reuse any UML editor or viewer tool for model visualization by generating the persistency format of the selected tool. We use the Ecore based implementation of UML2 as our internal model representation that gives us Papyrus-compatible [5] model persistency for free. Papyrus provides autolayout for diagrams.

Note, however, that automatically laid out diagrams are far from perfect, because the position of model elements and their relative distance in a hand-drawn diagram also carries information. For example, the class representing the most important concept is usually placed in the middle of a class diagram. Closely related classes are drawn close to each other, while marginal or technical associations can be long broken lines. Our plan is to extend the embedded language with constructs to define the relative placement of model elements. For example, one could specify that the `User` class has to be displayed next to the class `Machine`, on its right hand side. This is definitely an important future work in our project.

Figure 4 shows the Papyrus editor showing our example model. The tree-view of the model in the *Model Explorer* on the left hand side is automatically loaded when opening the model persisted by our API. The class diagram on the right-hand side can be created by manually dragging the classes from the Model Explorer onto the diagram.

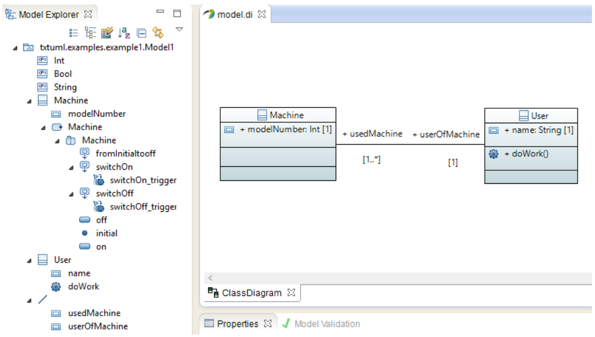


Fig. 4. The Papyrus editor showing the example model of Section 3

6 Translation

The toolchain of executable modeling is completed by the *model compiler* that turns a model into a program in a given implementation language (C++, Java, etc.) optimized towards a selected platform and runtime libraries. Our implementation includes a prototype model compiler generating C++. For example, let us show a fragment of the generated code for the example used throughout the paper:

```
struct Machine
{
  std::vector<User*> userOfMachine;
  enum state { state_Init, state_Off, state_On };
  state current_state;
  /* ... */
};
```

The association between the machine and its users results in the member called `userOfMachine` in the machine class. Since the multiplicity of this end of the association is `@Many`, a vector of pointers to user objects is generated. Note, that the compiler is free to choose a different collection type for optimization reasons. In particular, this member can be omitted if the association is never used to navigate from machines to users. The states of the machine are represented by the enumeration type `state` and the `current_state` member is responsible for storing the actual state in runtime.

Section 5 revealed that we use the Ecore-based UML2 implementation as the internal representation of the models. The model compiler's task is to query the model via the UML2 API and turn its elements C++ to code.

7 Implementation

7.1 Implementation of the execution semantics

Execution semantics of the model are captured by the implementation of our API, introduced in Section 3. The interesting bit of this implementation is the asynchronous communication between object instances. Any modeled class has to inherit from `ModelClass`. Its constructor creates a Java thread for all the instances[2]. Each thread has a mailbox of type `LinkedBlockingQueue<Signal>` that other instances can put signals in asynchronously. The threads run the following loop:

```
while (true) {
    Signal signal = mailbox.take();
    parent.processSignal(signal);
}
```

---

[2] This is the reason to use inheritance to mark classes belonging to the model.

That is, they are polling their mailboxes for incoming signals and propagate those to their object instances (`parent`) for processing. The `processSignal` method implements the runtime semantics of state machines. Note that the `take` method blocks the thread if the mailbox is empty, therefore no busy-waiting is involved. When the object instance is deleted, its thread is interrupted, leading to an `InterruptedException` that stops the loop above.

7.2 Building the internal representation of models

Both visualization and model compilation requires a representation of models that can be queried and transformed to graphics or text. In our case this representation is the Ecore-based implementation of UML2.

Using Java reflection, it is possible to get information about the static structure of Java code. For example, in case of the `Machine` class, we iterate through its attributes and member functions to find out the attributes and operations to be added to the model representation via the UML2 API. Then, the enclosed classes are examined, if they inherit from `State` or `Transition` in order to add the state machine of this class to the model.

Java reflection is unable to look into method bodies, but it is possible to call methods via the reflection API. However, simply calling the methods found eg. in states or transitions would not help in building model representation, since the instructions inside these methods implement the runtime semantics of the model. For this reason, we use AspectJ, an aspect-oriented extension to Java. For example, the following AspectJ code fragment replaces all method calls on instances of `ModelClass` with code that adds the corresponding call operation element to the model:

```
Object around(ModelClass target): call(* *(..)) && target(target) {
  return Importer.call(target, thisJoinPoint.getSignature().getName(),
                       thisJoinPoint.getArgs());
}
```

8 Summary

In this paper we presented *txtUML*, which is an executable UML language embedded in Java. Our main contributions are the following:

- A novel textual UML modeling method that makes model-level execution and debugging possible by reusing the Java runtime environment and Java debuggers.
- Application of Java reflection and AspectJ for creating model representation that can be used for visualization and model compilation.
- Prototype implementation of the framework.

Using a mainstream programming language provides easy integration to existing toolchains, while reusing its execution environment and debuggers reduces the implementation effort needed to create a framework for executable UML.

References

1. Epsilon project. `http://www.eclipse.org/epsilon/`
2. Executable Translatable UML Open Source Editor. `https://www.xtuml.org/`
3. MetaUML project. `http://metauml.sourceforge.net/old/`
4. Moka. `http://wiki.eclipse.org/Papyrus/UserGuide/ModelExecution`
5. Papyrus. `http://wiki.eclipse.org/Papyrus`
6. TextUML project. `https://github.com/abstratt/textuml`
7. The txtUML project. `http://txtuml.inf.elte.hu/`
8. Topcased migrates to PolarSys. `http://polarsys.org/topcased-migrates-polarsys`
9. Umple project. `http://cruise.eecs.uottawa.ca/umple/`
10. Yakindu Statechart Tools. `http://statecharts.org/`
11. Alf Open Source Implementation. `http://modeldriven.org/alf/` (2013)
12. Forward, A., Badreddin, O., Lethbridge, T.C.: Umple: Towards combining model driven with prototype driven system development. In: Rapid System Prototyping (RSP), 2010 21st IEEE International Symposium on. pp. 1–7. IEEE (2010)
13. Grönniger, H., Krahn, H., Rumpe, B., Schindler, M., Völkel, S.: Textbased modeling. In: 4th International Workshop on Software Language Engineering (2007)
14. Kirshin, A., Dotan, D., Hartman, A.: A uml simulator based on a generic model execution engine. In: MoDELS Workshops. vol. 4364, pp. 324–326 (2006)
15. MentorGaphics: BridgePoint xtUML tool. `http://www.mentor.com/products/sm/model_development/bridgepoint/`
16. Object Management Group: UML Human-Usable Textual Notation (HUTN), standard, version 1.0. `http://www.omg.org/spec/HUTN/` (2004)
17. Object Management Group: Action Language for Foundational UML (ALF), standard, version 1.0.1. `http://www.omg.org/spec/ALF/` (2013)
18. Object Management Group: Semantics of a Foundational Subset for Executable UML Models (fUML), standard, version 1.1. `http://www.omg.org/spec/FUML/1.1/` (2013)
19. Object Management Group: Unified Modeling Language (UML), standard in preparation, version 2.5 beta 2. `http://www.omg.org/spec/UML/2.5/Beta2/` (2013)
20. Object Management Group: Precise Semantics of UML Composite Structures (PSCS), standard in preparation, version 1.0 beta 1. `http://www.omg.org/spec/PSCS/1.0/Beta1/` (2014)
21. Paige, R.F., Kolovos, D.S., Rose, L.M., Drivalos, N., Polack, F.A.: The design of a conceptual framework and technical infrastructure for model management language engineering. In: Engineering of Complex Computer Systems, 2009 14th IEEE International Conference on. pp. 162–171. IEEE (2009)
22. Pontisso, N., Chemouil, D.: Topcased combining formal methods with model-driven engineering. In: Automated Software Engineering, 2006. ASE'06. 21st IEEE/ACM International Conference on. pp. 359–360. IEEE (2006)
23. Raistrick, C.: Model driven architecture with executable UML, vol. 1. Cambridge University Press (2004)
24. Shlaer, S., Mellor, S.J.: The Shlaer-Mellor method. Project Technology white paper (1996)
25. Steinberg, D., Budinsky, F., Merks, E., Paternostro, M.: EMF: Eclipse Modeling Framework. Pearson Education (2008)

MQT, an Approach for Runtime Query Translation: From EOL to SQL

Xabier De Carlos[1], Goiuria Sagardui[2], and Salvador Trujillo[1]

[1] IK4-Ikerlan Research Center, P. J.M. Arizmendiarrieta, 2 20500 Arrasate, Spain
`{xdecarlos,strujillo}@ikerlan.es`
[2] Mondragon Unibertsitatea, Goiru 2, 20500 Arrasate, Spain
`gsagardui@mondragon.edu`

Abstract. Managing models requires extracting information from them and modifying them, and this is performed through queries. Queries can be executed at the model or at the persistence-level. Both are complementary but while model-level queries are closer to modelling engineers, persistence-level queries are specific to the persistence technology and leverage its capabilities. This paper presents MQT, an approach that translates EOL (model-level queries) to SQL (persistence-level queries) at runtime. Runtime translation provides several benefits: (i) queries are executed only when the information is required; (ii) context and metamodel information is used to get more performant translated queries; and (iii) supports translating query programs using variables and dependant queries. Translation process used by MQT is described through two examples and we also evaluate performance of the approach.

Keywords: Model-Driven Development, Large-Scale Models, Runtime Query Translation, Model Queries

1 Introduction

Managing models encompasses tasks such as model validation, constraint checking, model analysis, etc. These tasks require executing queries over the models for getting information from them and also for modifying them. Model queries can be classified on two levels: model-level and persistence-level. Queries from both levels are complementary, but each level provides different benefits and limitations.

On the one hand, *model-level queries* are closer to modelling engineers since they are expressed in languages focused on interacting with models, independently of the persistence mechanism. Model-level query languages are for example Epsilon Object Language (EOL), Object Constraint Language (OCL), EMF Query, IncQuery, etc. Model-level query languages typically require the user to load the relevant models into memory first (e.g. using the `Resource` provided by EMF), before queries can be executed.

On the other hand, *persistence-level queries* are specific and dependent on a particular persistence mechanism. Some persistence-level query languages are for example: SQL for querying information persisted in relational databases; or MorsaQL [1] for querying models persisted using Morsa [2]. Persistence-level queries are typically executed directly over persisted models, without requiring to load model information first. Moreover, persistence-level languages leverage capabilities of persistence mechanisms for which they were created.

In this paper, we present the Model Query Translation layer (MQT), an approach that translates EOL queries (model-level) to SQL queries (persistence-level) at runtime. Runtime translation allows to translate imperative model-level queries that make use of variables and dependant queries. Moreover, runtime translation produces more performant queries through context and metamodel information.

The main contribution of this paper is MQT, a prototype that translates EOL queries to SQL. Using MQT, queries are translated and executed at runtime. We have evaluated it and the results show that the translation mechanism provided by MQT performs better (in terms of execution speed) than using the naive translation provided by Epsilon Model Connectivity Layer (EMC). Modification queries are not supported at this stage but we plan to add support for them in the future. Moreover, although currently MQT only translates EOL queries to SQL, in the future, we aim to extend it to support more model and persistence-level query languages.

The rest of the paper is organised as follows. In Section 2 we provide some background and motivation for this work. Section 3 presents MQT and illustrates the translation process through two examples. The approach is evaluated in Section 4, and Section 5 compares our approach with related work. We conclude the paper on Section 6 providing conclusions and directions for further work.

2 Background and Motivation

XMI is commonly used as a model persistence format in Eclipse Modelling Framework (EMF). Although other alternative file-based persistence solutions such as binary (supported by EMF) or JSON[3] exist, file-based persistence entails memory and performance problems with large models. When trying to solve scalability problems, most recent approaches [3,4,5,6] propose leveraging databases for large-scale model persistence. These approaches provide persistence-level query languages that leverage capabilities of these databases. This results in queries that are expressed at a low level of abstraction and tightly couples the queries with the specific model persistence mechanism used.

This scenario motivates us to provide a solution that automates query translation from persistence-agnostic model-level queries that are widely-used by modelling engineers to persistence-level queries that leverage the capabilities of the persistence layer to improve efficiency.

---

[3] Read more at `http://ghillairet.github.io/emfjson/`

3 MQT: Runtime Query Translation From EOL to SQL

Our work is focused on the implementation of a mechanism that allows querying models in a transparent way and using query languages closer to modelling engineers. Following these directions, we present MQT, an approach that provides automatic query translation from EOL, a model-level query language, to SQL, a persistence-level query language.

On the one hand, we have chosen EOL at model-level since it is an imperative OCL-like language that allows querying and modifying models of arbitrary modelling technologies. EOL provides interesting features such as: use of variables and methods, use of query chains and syntactic checking. Moreover, it is the base language of Epsilon Languages which provide different functionalities: model validation, model transformation, code generation, etc. On the other hand, we have chosen SQL at persistence-level since it is a structured and mature language widely used for performing queries in relational databases. However, it can be also used to query some NoSQL databases (e.g. using Unity JDBC[4] for running SQL queries against MongoDB databases).

We base our work on [7], where EOL is used to efficiently query large datasets stored on a single relational table. MQT supports translation of EOL query programs to SQL, and translated queries are only executed when the result is required. EOL is imperative and allows specifying variables and query chains. SQL does not support these features, which means that the expressive power of EOL and SQL is different. Consequently, the query translation cannot be total. Being so, in this case where the translated language (EOL) provides constructs that have no direct mapping in the target language (SQL), translation should be performed partially. The partial translation mechanism of MQT is based on the EMC, an API that provides abstraction facilities over modelling and data persistence technologies. To support the query translation, MQT prototype provides different classes that are described below:

- `MQTModel` It extends the `IModel` class provided by EMC and is the main class of the approach. Using this class MQT is able to interact with models conforming EMF in a uniform manner. In order to execute the SQL queries over databases where models are persisted, this class also implements a *jdbc* driver.
- `MQTResultObject` class. It implements the `IModelElement` interface of EMC and allows to work with each model element persisted within the database. This class translates, executes and get results of queries that ask about model elements and its features.
- `MQTResultSetList` class. This class translates, executes and get results of queries returning a list of `MQTResultObjects`.
- `MQTPrimitiveValueList` class. It implements the `IModelElement` interface and it is used to translate, execute and get results of SQL queries returning lists of primitive values (e.g. names).

---

[4] Read more at `http://www.unityjdbc.com/mongojdbc/mongo_jdbc.php`

Using these classes the approach is able to translate and execute at runtime EOL queries to SQL, and then get the results from a database where the queried model is persisted. However, the translation provided by EMC is **naive**: each query expression is translated and executed by the `MQTModel` and results of the SQL query are returned using `MQTResultObject`, `MQTResultSetList` or `MQTPrimitiveValueList`. Each query expression is translated and executed one-by-one. This occurs for example in the query `‘‘EClass.all.select(...)’’`: first `‘‘EClass.all’’` is translated, returning a `MQTResultSetList` class with the result. Then, the `‘‘.select(...)’’` part of the query is translated, but using the `MQTResultSetList`. This requires to execute a SQL query for each result of the list to check the condition of the select.

But the previously described translation mechanism does not exhibit a good performance when translating and executing complex queries over large models, since it executes a lot of SQL queries. To improve scalability within query translation, MQT provides a mechanism that adapts translated SQL queries at runtime. To support the adaptive query translation `MQTResultSetList` class implements the `IAbstractOperationContributor` interface. In this way, naive translation of more complex queries such as selects, collects or rejects is replaced by our own translation mechanism. Using this mechanism MQT is able to group related and dependant EOL queries within a single translated SQL query. Being so, MQT executes fewer SQL queries (they are more accurate) and are executed only when the results are required.

We now explain the runtime query translation mechanism, by demonstrating the execution of the translation of different EOL programs. To avoid introducing a custom metamodel, sample query programs are executed over models that conform to the Ecore metamodel. As we have chosen SQL persistence-level query language, we have provided a metamodel-agnostic mechanism that persists models in a relational database. Relational databases require the specification of a schema where the structure of the database is defined. However, it is important to note that this paper is focused on the translation of queries and not on the efficient persistence of large-scale models. In order to facilitate understanding of the query translation process, the data-schema of the database is described previously.

3.1 Data-Schema

As the schema of Figure 1 illustrates, table *Object* persists elements of the model. This table contains *ObjectID* (primary key) identifying each element in the model and the *classID* (foreign key) that stores ID of its meta-class. Meta-classes are stored within the *Class* table, where *ClassID* (primary key) and *Name* are stored for each one. *Feature* table persists ids (*FeatureID*) and *Name*s of all the existing attributes and references. Values of attributes and references (*FeatureID*) of each model element (*ObjectID*) are stored in the *AttributeValue* and *ReferenceValue* tables (primitive value in case of attributes and id and meta-class in case of references). These five tables are enough for persisting the information stored in the models. However, sometimes adding duplicated information has a significant

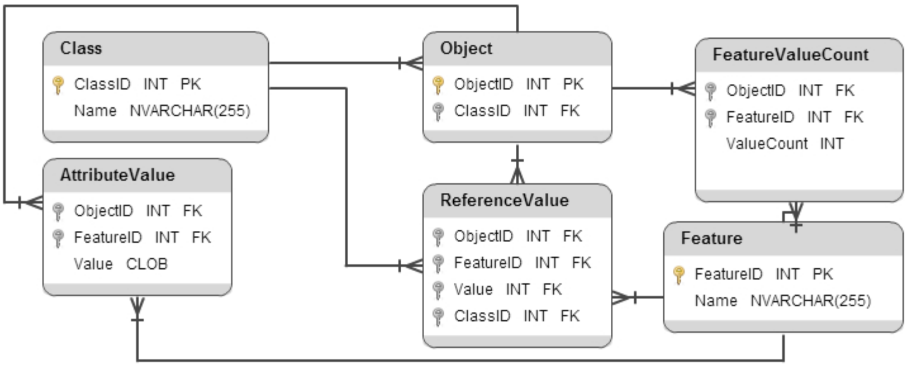


Fig. 1: Specified metamodel-agnostic database schema.

performance impact on the execution since the information can be used to get more effective translated queries (in terms of query execution speed). This is the case of *FeatureValueCount* table containing how many values each feature of each element has. Information of this table is used to get more effective translated queries: for example if *ValueCount* is 0, the feature do not exist for the element, and consequently, there is no need to query value of the feature; if *ValueCount* is X, it can be used to add `‘‘LIMIT X’’` expression within the translated SQL query.

To improve query execution speed, MQT loads some information of the database in the memory during query translation and execution. This is the case of `FeatureIDCache` and `ClassIDCache` (loading ID and name of the existing features and classes).

3.2 Translation Example 1: Simple queries

The first example, shown in Listing 1.1, illustrates a querying program which: retrieves all model elements that are instances of `EClass` and assigns only the first element of the list to the `class` variable (line 1); then, the program prints the value of the name attribute of the selected element (line 2); and finally, the program iterates elements that are referenced through the `eSuperTypes` reference of the selected element (line 3), printing the name of each one of them (line 4).

In the following paragraphs, we explain how MQT translates and executes these queries.

```
1  var class = EClass.all.first();
2  class.name.println();
3  for (superClass in class.eSuperTypes)
4    superClass.name.println();
```

Listing 1.1: Sample EOL program getting instances and attributes/references.

- **Line 1:** `EClass.all.first()`. This first query is divided in two parts. The first part is `EClass.all` and it creates a new instance of `MQTResultSet` adding the information required to query for all model elements that are instances of `EClass`. As such, behind the scenes, `MQTResultSet` executes the following SQL query to retrieve the list of matching elements from the database: `SELECT ObjectID FROM Object WHERE ClassID=?` The parameter of the query is `ClassID` obtained from memory using the previously introduced `ClassIDCache`. Then, the second part is executed (*first()* expression), returning only the first element (instance of `MQTResultObject`) of the `MQTResultSetList`, and it is stored in the *class* variable.
- **Line 2:** `class.name`. This query expression prints the name of the previously selected `MQTResultObject`. This is provided by a method implemented within the `MQTResultObject` (`getValue(feature)`) that retrieves attribute and reference values of the related model element from the database. In this case, the method executes the following query for getting the value of the name attribute: `SELECT Value FROM AttributeValue WHERE ObjectID = ? and FeatureID = ? LIMIT 1`. The query parameters are the `ObjectID` known by the `MQTResultObject` and the `FeatureID` obtained from memory (`FeatureIDCache`).
- **Line 3:** `for (superClass in class.eSuperTypes)`. This statement is similar to the previous but instead of returning an attribute value, a list of *ResultObject*s is returned and then iterated. The executed query is the following: `SELECT Value FROM ReferenceValue WHERE ObjectID = ? and FeatureID = ?`.
- **Line 4:** `superClass.name`. The execution logic of this statement is similar to the `class.name` statement.

3.3 Translation Example 2: Complex queries

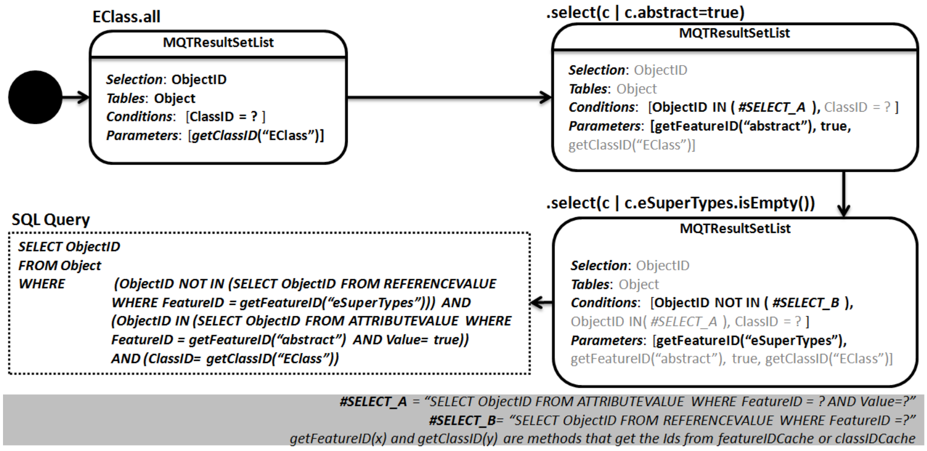
Listing 1.2 shows another EOL program with more complex model element selection queries. In this example, the program first selects all the abstract EClasses without superclasses, and then computes the number of model elements that satisfy these conditions. If at least one element satisfies these conditions, the program prints the name of the first model element of the list.

```
var list = EClass.all.select(c|c.abstract=true)
var list2 = list.select(c|c.eSupertypes.isEmpty());
if(list2.size() > 0)
   list2.first().name.println();
```

Listing 1.2: Sample EOL program with selection.

Figure 2 illustrates the translation mechanism used for translating complex queries from Listing 1.2. Following, a more detailed description is provided:

- `EClass.all`. Creates a new instance of *MQTResultSetList* adding the information required to query all the model elements that are of the *EClass* meta-type.



EClass.all — MQTResultSetList

Selection: ObjectID
Tables: Object
Conditions: [ClassID = ?]
Parameters: [*getClassID*(“EClass”)]

.select(c | c.abstract=true) — MQTResultSetList

Selection: ObjectID
Tables: Object
Conditions: [ObjectID IN (*#SELECT_A*), ClassID = ?]
Parameters: [getFeatureID(“abstract”), true, getClassID(“EClass”)]

.select(c | c.eSuperTypes.isEmpty()) — MQTResultSetList

Selection: ObjectID
Tables: Object
Conditions: [ObjectID NOT IN (*#SELECT_B*), ObjectID IN(*#SELECT_A*), ClassID = ?]
Parameters: [getFeatureID(“eSuperTypes”), getFeatureID(“abstract”), true, getClassID(“EClass”)]

SQL Query

```
SELECT ObjectID
FROM Object
WHERE        (ObjectID NOT IN (SELECT ObjectID FROM REFERENCEVALUE
             WHERE FeatureID = getFeatureID(“eSuperTypes”))) AND
             (ObjectID IN (SELECT ObjectID FROM ATTRIBUTEVALUE WHERE
             FeatureID = getFeatureID(“abstract”) AND Value= true))
             AND (ClassID= getClassID(“EClass”))
```

#SELECT_A *= “SELECT ObjectID FROM ATTRIBUTEVALUE WHERE FeatureID = ? AND Value=?”*
#SELECT_B*= “SELECT ObjectID FROM REFERENCEVALUE WHERE FeatureID =?”*
getFeatureID(x) and getClassID(y) are methods that get the Ids from featureIDCache or classIDCache

Fig. 2: Example of the runtime translation of the query from EOL to SQL.

- `select(c|c.abstract=true)`. This second expression completes the previous one. Being so, the same instance of `MQTResultSetList` is completed by adding a new condition and two parameters to get only the model elements that have the abstract feature with the true value.
- `select(c|c.eSuperTypes.isEmpty())`. This select expression completes more the `MQTResultSetList` instance. It adds a new condition and parameter to get only the model elements that do not have `eSuperTypes`.
- Finally, the information that has been added during the previous steps is used by the `MQTResultSetList` to get the translated SQL query.

MQT only executes the translated SQL query over the database when the results are needed. In the case of the previous example (Listing 1.2), the instance of `MQTResultSetList` executes the translated SQL in the line 3, where the size value needs be returned. Once the query described on Figure 2 is executed, `MQTResultSetList` instance obtains the results. Then `MQTResultSetList` provides size through a method that counts the quantity of the returned elements. In the case of line 4, `MQTResultSetList` uses same results but it returns only the first element of the list.

4 Evaluation

To evaluate MQT, we have executed an EOL query program over five models of different sizes (from 45MB to 403MB). All experiments have been executed using an Intel Core i7-3520M CPU at 2.90 GHz with 8GB of physical RAM running Windows 7 SP1 64bit, JVM 1.7.0 and the Eclipse Kepler SR1 distribution configured with 2GB of maximum heap size. Models have been created using MoDisco's Java Discoverer [8] and they specify source code of different Java plug-ins. These models have been persisted in a relational database with the previously described metamodel-agnostic schema and using the H2[5] DBMS (version 1.3.168). We have specified an EOL Query Program based on the GraBats'09 reverse engineering contest, where the query identifies singleton classes within source code of Java plug-ins specified by the models. We have used GraBats'09 because is widely used by the community on the studies related to model persistence.

EOL Query program has been executed 100 times over each model and using the naive translation of EMC and the MQT translation (previously explained on Section 3). Table 1 shows information of the models used during experimentations: *size*, number of *objects*, number of *methods* and number of *singleton classes* that they contain.

Table 1: Singleton query program execution over different models.

	Size	Objects	Methods	Singl. Classes	Naive trans.	MQT trans.
M1	45MB	165741	5366	9	185ms	13ms
M2	72MB	330761	8129	8	302ms	11ms
M3	212MB	875988	11393	6	676ms	10ms
M4	327MB	1343207	15386	0	950ms	3ms
M5	403MB	1566890	19366	0	1243ms	1ms

Average of the query translation and execution times obtained during experimentations are illustrated on *Naive translation* and *MQT translation* columns. As is shown on the table, with naive translation, model size has great impact on the required time for executing the query program and it increases as the size of the model increases. However, using MQT translation model size has less impact over the execution time. We can conclude with these results that MQT provides more scalability if queries are executed over large-models.

To assess the performance of MQT, we executed the same query against XMI models (provided by EOL) and obtained the same results. We have also analysed the execution time spent on the query translation: (M1) 3.35ms; (M2) 4,51ms; (M3) 0.77ms; (M4) 0.8ms; and (M5) 0.64ms. From these results, we have concluded that as the translation time is only few milliseconds it does not imply a temporary overload.

5 Related Work

Several solutions have been proposed in terms of generation of queries based on OCL-like languages. [9] proposes an approach focused on generating SQL queries from invariants specified using OCL. The approach allows mapping Unified Modelling Language (UML) models to other data schemas like databases

---

[5] More information about H2 at `http://www.h2database.com/`

and then generating queries that allow to evaluate invariants using SQL. [10] describes an approach that generates views using OCL constraints, and then uses these view to check the integrity of the persisted data. The approach has been implemented in OCL2SQL[6], a tool that generates SQL queries from OCL constraints. A similar approach for integrity checking is proposed in [11]. While these approaches are focused on translating OCL constraints into SQL queries at compile-time, our approach generates SQL queries from OCL-like expressions (EOL) at runtime. Comparing with compilation-time translation, main benefits of runtime translation are: (i) translated queries are executed only when the information is required; (ii) context and metamodel information can be used during query translation; and (iii) it supports query chains and variables within the query program.

[12] presents SPARQLAS, an SPARQL-like query syntax that is translated to SPARQL and then executed against OWL knowledge base, using results as input for OCL queries. Using SPARQLAS, queries are executed using SPARQL (persistence-level) and then query results are the input of OCL queries (model-level). By contrast, our approach translates queries from EOL (model-level) to SQL (persistence-level) and then executes the obtained SQL queries.

6 Conclusions and Further Work

In this paper we have presented MQT, an approach that translates at runtime and automatically model queries specified using EOL (model-level query language) to SQL (persistence-level query language). Main benefits of the runtime translation are: (i) on-demand execution of translated queries; (ii) queries are translated and adapted at runtime and metamodel and context is used to get more effective translated queries; (iii) it allows to execute query programs with variables and dependant queries.

We have evaluated our approach, concluding that with the runtime translation and adaptation of queries, MQT provides a scalable solution to query large-models. However, for the future, we plan to perform a more complete evaluation that analyses the impact of the characteristics of the used models and queries.

Presented prototype supports all the EOL expressions that obtain information from models, but modification expressions are not supported. For a next version, we plan to extend the approach with support for: (i) modification queries; (ii) additional model-level query languages (e.g. OCL); and (iii) additional persistence-level query languages (e.g. Cypher). This will provide a solution that allows engineers to write high-performance queries in a model-level query language without worrying about the model persistence format. Regarding this point, an open issue to be analysed is how to provide extensibility to facilitate the integration of query languages at both sides.

---

[6] Read more at `http://dresden-ocl.sourceforge.net/usage/ocl22sql/`

Acknowledgments

Authors wish to thank Dr. Dimitris Kolovos for his help on this work. This work is partially supported by the EC, through the Scalable Modelling and Model Management on the Cloud (MONDO) FP7 STREP project (#611125).

References

1. Pagán, J.E., Molina, J.G.: Querying Large Models Efficiently. Information and Software Technology **56**(6) (2014) 586 – 622
2. Espinazo Pagán, J., Sánchez Cuadrado, J., García Molina, J.: Morsa: A Scalable Approach for Persisting and Accessing Large Models. In Whittle, J., Clark, T., Khne, T., eds.: Model Driven Engineering Languages and Systems. Volume 6981 of Lecture Notes in Computer Science. Springer Berlin Heidelberg (2011) 77–92
3. Eike Stepper: CDO Model Repository Overview. `http://www.eclipse.org/cdo/documentation/` Accessed March 17, 2014.
4. Pagán, J.E., Cuadrado, J.S., Molina, J.G.: A Repository for Scalable Model Management. Software & Systems Modeling (2013) 1–21
5. Benelallam, A., Gómez, A., Sunyé, G., Tisi, M., Launay, D.: Neo4EMF, a Scalable Persistence Layer for EMF Models. In: ECMFA- European conference on Modeling Foundations and applications, York, UK, Royaume-Uni, Springer (July 2014)
6. Scheidgen, M.: Reference Representation Techniques for Large Models. In: Proceedings of the Workshop on Scalability in Model Driven Engineering. BigMDE '13, New York, NY, USA, ACM (2013) 5:1–5:9
7. Kolovos, D.S., Wei, R., Barmpis, K.: An Approach for Efficient Querying of Large Relational Datasets with OCL-based Languages. In: XM 2013–Extreme Modeling Workshop. (2013) 48
8. Bruneliere, H., Cabot, J., Jouault, F., Madiot, F.: MoDisco: A Generic and Extensible Framework for Model Driven Reverse Engineering. In: Proceedings of the IEEE/ACM International Conference on Automated Software Engineering. ASE '10, New York, NY, USA, ACM (2010) 173–174
9. Heidenreich, F., Wende, C., Demuth, B.: A Framework for Generating Query Language Code from OCL Invariants. ECEASST **9** (2008)
10. Demuth, B., Hussmann, H., Loecher, S.: OCL as a Specification Language for Business Rules in Database Applications. In Gogolla, M., Kobryn, C., eds.: UML 2001 The Unified Modeling Language. Modeling Languages, Concepts, and Tools. Volume 2185 of Lecture Notes in Computer Science. Springer Berlin Heidelberg (2001) 104–117
11. Marder, U., Ritter, N., Steiert, H.: A DBMS-Based Approach for Automatic Checking of OCL Constraints. In: Proceedings of Rigourous Modeling and Analysis with the UML: Challenges and Limitations. OOPSLA (1999)
12. Parreiras, F.S.: Semantic Web and Model-driven Engineering. John Wiley & Sons (2012)

Incremental Checking of OCL Constraints through SQL Queries

Xavier Oriol and Ernest Teniente

Department of Service and Information System Engineering
Universitat Politècnica de Catalunya – BarcelonaTech
`{xoriol,teniente}@essi.upc.edu`

Abstract. We propose a new method for efficiently checking OCL constraints by means of SQL queries. That is, an OCL constraint is satisfied if its corresponding SQL query returns the empty set. Such queries are computed in an incremental way since, whenever a change in the data occurs, only the constraints that may be violated because of such change are checked and only the relevant values given by the change are taken into account. Moreover, the queries we generate do not contain nested subqueries nor procedures. In this way, we take advantage of relational DBMS capabilities and we get an efficient check of OCL constraints.

Keywords: OCL, Constraints Checking, SQL

1 Introduction

A conceptual schema is the description of an Information System in terms of the data it should contain and the operations available to users to modify such data [1]. To define conceptual schemas, the Object Management Group (OMG) has defined the UML/OCL standards [2,3]. Broadly speaking, UML is used for specifying a class diagram, i.e. the structure of the data, and OCL for stating the conditions (i.e. the constraints) that should always be satisfied by the data.

We aim at defining an efficient method to perform integrity checking of OCL constraints. That is, to efficiently check whether the OCL constraints of the schema are satisfied by the data contents. Such problem arises in several situations like conceptual schema execution in animation tools (like USE [4]) or checking whether a model satisfies the constraints defined in its metamodel in MDA [5]. Unfortunately, there are not efficient OCL checkers able to deal with medium-large scenarios [6].

One way to efficiently check OCL constraints is aimed at reducing such problem to check the emptiness of some SQL query [7]. Intuitively, given an OCL constraint, we can build an SQL query that returns all instances that violate it. Thus, the OCL constraint is satisfied if and only if the SQL query is empty. In this way, we benefit from all optimization techniques of current DBMS for query answering. Moreover, since SQL is widely used for storing data from constrained domains, bringing an efficient method for checking constraints based on SQL might be integrated in current industrial systems without crossing technologies.

To our knowledge, there are two implemented tools that perform such translation: OCL2SQL [8] and MySQL4OCL [9]. However, their translation should be further optimized to scale up for efficient integrity checking in large scenarios. Mainly, because whenever a change in the data occurs (e.g. an insertion of a new instance), the whole query is recomputed, when it probably just need to check whether the updated data should appear as a result of the query.

The main goal of this paper is to overcome the previous drawback by proposing a translation from OCL constraints to SQL queries which allows to compute them incrementally by a relational DBMS. This is achieved by generating SQL queries which are only recomputed when the change applied to the data may violate their associated constraint and such that, whenever the computation is performed, only the relevant values given by the update are taken into account. In addition, our generated queries do not nest subqueries nor procedures.

Our method starts by applying the automatic translation of OCL Constraints into Event Dependency Constraints (EDCs), as defined in [10]. An EDC is a logic formula which states when some structural events (i.e. insertions or deletions of data) may cause the violation of a constraint. In terms of logics, an EDC is a conjunctive query with negated base atoms and built-in literals (i.e. arithmetic comparisons). So, an EDC has the form: $l_1 \wedge ... \wedge l_n \wedge bil_1 \wedge ... \wedge bil_m \rightarrow \perp$, where each l_i is a literal representing an instance, an instance insertion or an instance deletion; and bil_j is a built-in literal. For example, the EDC: $user(X) \wedge \iota userAge(X, Age) \wedge Age < 18 \rightarrow \perp$ states that a constraint is violated if there is a user X in the data and we insert some Age below 18 to this user X.

From this point, we define an inductive translation from EDCs to SQL. Broadly speaking, l_1 gives the initial table to start the *FROM* clause, then, each l_i is joined with the other tables by a cross join (i.e. a Cartesian product), inner join or anti join depending on the binding of the variables of l_i and its positive/negative sign. Any bil_j is directly placed in the *WHERE* clause.

Literals l_i representing an instance insertion or deletion, i.e. a change taking place over the data, provide the key for incrementality. Each such l_i is translated as an SQL join from a table containing such insertions/deletions to the rest of tables translated from the other EDC literals. When a user wants to apply an update, this update is first inserted in the auxiliary tables and, once the method has ensured that it does not cause any integrity constraint violation, the update is applied to the real tables and the auxiliary tables are emptied. Such process may be automatized by means of a DB controller.

Since each SQL query contains at least one of such literals, such query computations will always be tied to the data changes. For instance, in the previous EDC, $\iota userAge(X, Age)$ joins the query with the insertion of an age to a user. Thus, if we do not insert any age for any user, the join will return the empty set and no more evaluation will be needed. On the contrary, if there is some insertion of an age to a user, the join will retrieve only the affected user(s) from which to continue the query computation. Note that such evaluation is better than retrieving all the users of the database and then perform the convenient checks for each one for any kind of data update.

As a result, we get some SQL queries that are empty if and only if the OCL constraints are satisfied. Such queries perform incrementally and, in addition, avoid the use of nested queries/procedures. As a trade-off, the expressiveness of OCL is limited to the fragment of OCL translatable to EDCs.

To show the benefits of our method, we have performed some experiments to compare the efficiency of our translation with the translations given by OCL2SQL and MySQL4OCL. In such empirical study, we show that whereas our approach is capable to check the integrity of a set of constraints in at most 3 seconds per constraint regarding scenarios with $5{\cdot}10^6$ instances and $5{\cdot}10^4$ updates, OCL2SQL and MySQL4OCL could not check some invariants after one hour of execution.

2 Related Work

To our knowledge, there are two tools implementing an OCL to SQL translation: OCL2SQL[8] and MySQL4OCL[9]. We review both of them separately. In addition, we review the research on incremental OCL integrity checking, some work based on translating OCL to graph patterns, and some other based on translating FOL based constraints to incremental SQL queries.

OCL2SQL. OCL2SQL is a component of the OCL Dresden Toolkit for translating OCL constraints to SQL views [8]. The idea is that such SQL views are empty if and only if all the constraints are satisfied. The bases for such translation were early established in [7], however, since such bases were not defined inductively, it is hard to realize which OCL subset does it deal with. In any case, the main drawback that we find in it is the non-incrementality of their checks.

MySQL4OCL. MySQL4OCL is a tool for translating OCL expressions to MySQL queries [9]. The translation is tied to MySQL because it uses some procedures for translating iterator expressions of OCL (e.g. *forAll*, *exists*, *select*, etc). In this way, they deal with a broad subset of OCL. Moreover, they can also deal with the three valuated logic of OCL (i.e. true, false, null). However, and similarly to OCL2SQL, the translation of the constraints is not incremental and thus, we do not know which queries should be recomputed and which is the relevant data to take into account to check the data integrity after some update.

Incremental OCL Integrity Checking. There are already some proposals on how to incrementally check OCL constraints [11,12,13]. Applying such methods it is possible to realize which constraints should be checked and which is the relevant data to analyse after some data updates. Nevertheless, as far as we know, none of them have been adapted to work with databases on its behind.

OCL translation to graph patterns. The work of [5] consists in translating OCL constraints into graph patterns to benefit from its incremental capabilities. Such proposal relies on the intensive usage of memory since it is not intended to work with databases, but from data kept in memory. To overcome such difficulty, in [14] there is a proposal implementing the incremental graph patterns approach using some distributed databases. However, it seems that such databases are just implementing the persistence and the data is still kept (and queried) in memory.

FOL to incremental SQL queries. Decker proposed a method to translate constraints specified in a logic notation (different from OCL) into incremental SQL queries [15]. In his proposal, constraints are translated into queries that are invoked by triggers when a change over the data may cause a violation of a constraint. This is the way efficiency is achieved. However, and despite the difference on the language used to specify the constraints, we are not aware of any implementation of this approach that allow us to compare its efficiency with ours in the experiments we have performed.

3 Translating OCL Constraints to SQL Queries

In the following, we first define a conceptual schema with some OCL constraints that will be used as a running example to illustrate our method. Then, we show the representation of the OCL constraints in the form of EDCs. Such EDCs adds the incrementality capabilities to constraint checking. Finally, we give an inductive translation from such EDCs to SQL queries to perform the incremental checks of the constraints by means of relational DBMS technology.

3.1 An Illustrative Example

Consider the example in Fig. 1 of a message service application. In this class diagram, a *user* sends *messages* to some *conversation groups*. There are two kinds of conversation groups: *pairs*, that is, two simple users sending messages to each other; and *groups*, that is, two or more users formally grouped since some creation date. As expected, a group has some users as *members* and also one user as *owner*.

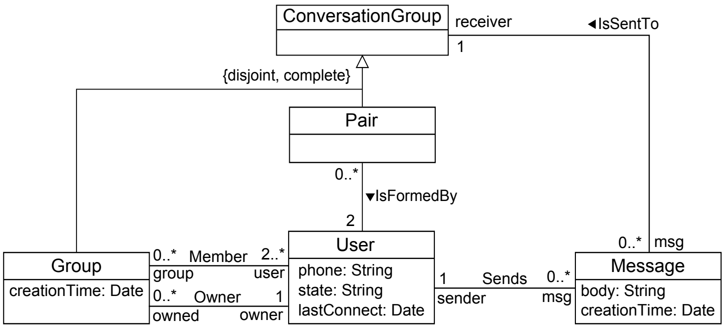


Fig. 1. Class diagram of a instant message service

OCL Constraints in Fig. 2 states some constraints that should always be satisfied by the data of such schema. Concretely, *MessagesInAPairBelongToPair* and *MessagesInAGroupBelongToGroup* state that the messages received by any pair/group are sent by the members of such pair/group. *UserIsMemberOfOwnedGroups* states that the owner of a group is also a member of such group and finally, *MessagesOfGroupAreSentAfterItsCreation* states that any messages sent to a group has to be created after the creation of the group.

Note that we have deliberately omitted the identifier constraints of the classes (e.g. *User*.`allInstances()->isUnique`(*phone*)) because they can be easily well treated by SQL primary keys on tables.

```
context Pair inv MessagesInAPairBelongToPair:
self.user->includesAll(self.msg.sender)

context Group inv MessagesInAGroupBelongToGroup:
self.user->includesAll(self.msg.sender)

context User inv UserIsMemberOfOwnedGroups:
self.group->includesAll(self.owned)

context Group inv MessagesOfGroupAreSentAfterItsCreation:
self.msg->forAll(e|e.creationTime > self.creationTime)
```

Fig. 2. OCL Constraints for the previous class diagram

Given a translation of the UML class diagram into SQL tables, our goal is to translate each OCL constraint into an SQL query in such a way that the constraint is not violated by an update (i.e. a change over the data) if and only if its corresponding SQL query is empty. For this purpose, we will first represent the violations of OCL constraints by means of EDCs and, then, obtain the SQL queries from this intermediate representation.

3.2 The EDC Representation of an OCL Constraint

The starting point of our work is the Event-Dependency Constraints (EDC) representation of the OCL constraint violations. The translation from OCL to EDCs can be fully automatized using the method described in [10]. For the sake of self-containment we review the formal notion of EDC below.

An EDC is a logic formula that states when some structural events (i.e. insertions or deletions of data) may cause the violation of a constraint. In terms of logics, an EDC is a conjunctive query with negated base atoms and built-in literals (i.e. arithmetic comparisons). That is, it has the form $l_1 \wedge ... \wedge l_n \wedge bil_1 \wedge ... \wedge bil_m \rightarrow \bot$ where each l_i is a literal representing either an instance, an instance insertion or an instance deletion; and bil_j is a built-in literal. Such built-in-literals are optional. In addition, EDCs are *safe clauses*, that is, any variable appearing in a negated or built-in literal, also appears in a positive literal.

For instance, the EDCs representation of the *UserIsMemberOfOwnedGroups* OCL constraint is:

$$\neg owner(G,U), \iota owner(G,U), \neg member(G,U), \neg\iota member(G,U) \rightarrow \bot \tag{1}$$

$$\neg owner(G,U), \iota owner(G,U), member(G,U), \delta member(G,U) \rightarrow \bot \tag{2}$$

$$owner(G,U), \neg\delta owner(G,U), member(G,U), \delta member(G,U) \rightarrow \bot \tag{3}$$

Note that an OCL constraint is written into more than one EDC since each EDC defines a different combination of events that may lead to the violation of the constraint. Concretely, Rule 1 above states that *UserIsMemberOfOwnedGroups* will be violated if we insert a user U as the owner of group G, where U is not a member of G and without inserting U as a member of G. Rule 2 identifies a violation of the same constraint when we insert U as the owner of G while deleting U as a member of G. Finally, Rule 3 indicates a violation when we delete U as a member of G without deleting U as the owner of G.

The length of an EDC is directly proportional to the length of the OCL constraint, but the number of EDC rules we get for an OCL constraint is exponential to the number of navigation steps of it since there is an exponential number of different ways to violate a constraint and each EDC captures one. To avoid such situation, we could take out the common factor of EDCs (e.g. rules 2 and 3 could be summarized in one rule using disjunctions). In this way, we would achieve a behavior similar to the TREAT algorithm [16] where joins are performed using the union between a table and its new insertions. However, in TREAT, each constraint is evaluated as many times as tables containing new instances are accessed in the definition of the constraint. The efficiency comparison of our proposal and TREAT is left out for further work since, as far as we know, there is no tool implementing TREAT in SQL for OCL constraints.

It is worth saying that the current translation from OCL to EDCs only deals with a fragment of OCL. However, such fragment is expressive enough to deal with a superset of the constraints that can be specified with the constraint patterns defined in [17], which have been shown to be useful for defining around the 60% of the integrity constraints found in real schemas. The grammar of the OCL constraints translatable into EDCs can be found in [10].

3.3 From the UML Class Diagram to SQL Tables

Before translating the EDCs into SQL queries, we need to translate the UML class diagram into SQL tables. In our example, we will suppose that each UML class/association has been translated into a different SQL table. Thus, the signatures of the tables obtained from the class diagram of Fig. 1 are the following:

Pair(id)
Owner(user, group)
Sends(user, message)
IsFormedBy(pair,user)
ConversationGroup(id)
Member(user, group)
Message(id, body, time)
Group(id,creationTime)
User(id,phone,state,lastConnect)
IsSentTo(conversgroup,message)

3.4 Translating EDCs to SQL Queries

We define now an inductive translation from EDCs to SQL queries based on the EDC length. The translation is composed by two functions: one for computing the *from* clause and another for computing the *where* clause. Regarding the *select* clause, we select the column that represents the id of the instances violating the constraint (i.e. the *self* OCL instances for which the invariant evaluates to false).

Therefore, the translation of an EDC to an SQL query is given by the pattern:

$$\texttt{SELECT}\ \mathit{sqlColumn}(\mathit{edcVariable}(''\mathit{self}''))$$
$$\texttt{FROM}\ \mathit{fromTransl}(\mathit{EDC})$$
$$\texttt{WHERE}\ \mathit{whereTransl}(\mathit{EDC})$$

Where *sqlColumn* returns the SQL column name corresponding to the given EDC variable. The EDC variable in which we are interested is the one corresponding to the OCL *self* variable, so, we use the function *edcVariable* with the parameter "self" to obtain it. In case that there is no "self" variable in the OCL constraint (e.g. the constraint may be like *Class.allInstances()->forAll(e|...)*), other options should be considered like selecting the column/s corresponding to the OCL iterator variable/s (e.g. the previous e variable).

Since EDCs use some literals to represent the insertion/deletion of instances, we assume that our database schema contains also some public auxiliary tables in which we temporally store the instances that are being inserted/deleted. Thus, such literals are mapped to those auxiliary tables.

For instance, EDC 2 uses the literal $\iota owner$. Such literal is mapped to a new auxiliary SQL table *ins_owner* where we temporary write the insertions of instances of *owner* we are applying to the data.

Finally, recall that an EDC has the form $l_1 \wedge ... \wedge l_n \wedge bil_1 \wedge ... \wedge bil_m \rightarrow \bot$. Without loss of generality, we assume that all negated literals are placed in the end of the formula (i.e. from some l_i to l_n); and that all terms of a literal l_j are variables with different names. Such condition can be ensured by replacing some terms for new fresh variables and binding such variables to its actual terms with new built-in literals (e.g. *P(X,1)* would be translated to $P(X,Y) \wedge Y = 1$).

Translation Base Case In the base case, the EDC is composed by just one literal l_1. Such literal is necessarily positive to ensure the safeness of the clause. Then, the translation is as follows:

$$fromTransl(l_1) = sqlTable(l_1)$$
$$whereTransl(l_1) = \emptyset$$

Translation Inductive Case In the inductive case, the EDC has the form $L \wedge l_i$ or $L \wedge bil_i$, where L is a non-empty list of literals, l_i is a positive or negated literal, and bil_i a built-in literal.

In the first case, if l_i is positive, the translation consists in an inner join between the translations of l_i and L joining the columns corresponding to their bound variables. If no variable is bound, then, a cross join (i.e., a Cartesian product) is performed instead. Thus, the translation is as follows:

$$fromTransl(L \wedge l_i) = fromTransl(L)\ \texttt{JOIN}\ sqlTable(l_i)\ \texttt{ON}\ (binding(L, l_i))$$
$$whereTransl(L \wedge l_i) = whereTransl(L)$$

Where $binding(L, l_i)$ is a function that returns the column joins according to the variables of l_i that are bound to L.

If l_i is negative, the translation consists in performing an anti join to get those tuples of L that do not join l_i. For performing the anti join, we use a left join and check the joined columns to be *null*. Thus, the translation results in:

$$fromTransl(L \wedge l_i) = fromTransl(L)\ \texttt{LEFT JOIN}\ sqlTable(l_i)\ \texttt{ON}\ (binding(L, l_i))$$
$$whereTransl(L \wedge l_i) = whereTransl(L)\ \texttt{AND}\ columnName(l_i, 1)\ \texttt{IS NULL}$$

Where *columnName*(l_i, 1) returns the name of the first SQL column of the SQL table corresponding to l_i. Such column is used to check whether the joined columns resulted into *null*.

Also, notice that $binding(L, l_i)$ will not be empty because all the variables of l_i are necessarily bound to L since any EDC satisfies the *safeness* property.

Finally, the translation when we deal with a built-in literal bil_i is as follows:

$$fromTransl(L \wedge bil_i) = fromTransl(L)$$
$$whereTransl(L \wedge bil_i) = whereTransl(L) \texttt{ AND } sqlComparison(L, bil_i)$$

Where *sqlComparison* is a function that translates the bil_i into the SQL syntax. I.e. it changes the variables of bil_i for the corresponding SQL column names of the L variables they are bound to. Note that all variables of bil_i are bound to L because the *safeness* property.

Translation example To illustrate our translation, we show how the EDC in rule 2 is specified as an SQL query. First of all, we sort the EDC literals to move the negated ones to the end of the formula, thus obtaining the following EDC':

$$\iota owner(G, U), member(G, U), \delta member(G, U) \neg owner(G, U) \rightarrow \bot$$

Next, by applying the translation we have just defined, we get:

```
SELECT T0.user
FROM ins_owner AS T0
   JOIN Member AS T1 ON (T1.group = T0.group AND T1.user = T0.user)
   JOIN del_Member AS T2 ON (T2.group = T0.group AND T2.user = T0.user)
   LEFT JOIN Group AS T3 ON (T3.group = T0.group AND T3.group = T0.group)
WHERE T3.group IS NULL
```

Lastly, and since violations of an OCL constraint are specified by means of several EDCs, the final SQL query we obtain to check the OCL constraint is given by the SQL union of all the queries we have obtained from each EDC.

4 Experiments

We have conducted an experiment to illustrate the scalability improvement provided by our SQL queries as compared to OCL2SQL[8] and MySQL4OCL[9]. In this experiment, we show that we can scale up to scenarios with $5*10^6$ instances with $5 \cdot 10^4$ updates to check a set of OCL constraints whereas OCL2SQL and MySQL4OCL can not deal with some of them after 1 hour of execution. Since the time required to check a constraint is independent from the others, we stayed at a reduced number of constraints without altering the relevance of our results.

Given the schema of a message service application in our example, the conducted experiment consisted into adding some new message instances to some conversation groups in several randomly generated database states of increasing size. Then, we checked all the constraints of the example by means of the queries generated by MySQL4OCL, OCL2SQL and our incremental approach.

Table 1. Time results comparison with OCL2SQL and MySQL4OCL

	5*10^3	5*10^4	5*10^5	5*10^6
1st Const. - MySQL4OCL	0.71 s	7.45 s	58.0 s	> 1 h
1st Const. - OCL2SQL	0.17 s	0.40 s	3.37 s	> 1 h
1st Const. - inc.	0.09 s	0.13 s	0.46 s	2.63 s
2nd Const. - MySQL4OCL	0.70 s	7.04 s	58.9 s	> 1 h
2nd Const. - OCL2SQL	0.15 s	0.54 s	3.71 s	> 1 h
2nd Const. - inc.	0.10 s	0.12 s	0.34 s	2.38 s
3rd Const. - MySQL4OCL	0.60 s	2.06 s	17.0 s	223 s
3rd Const. - OCL2SQL	0.11 s	0.17 s	0.51 s	42.15 s
3rd Const. - inc.	0.09 s	0.09 s	0.10 s	0.10 s
4th Const. - MySQL4OCL	1.94 s	15.0 s	126 s	> 1 h
4th Const. - OCL2SQL	0.11 s	0.25 s	1.72 s	> 1 h
4th Const. - inc.	0.09 s	0.09 s	0.24 s	1.63 s

The randomly generated data consisted in N users who randomly had already sent $N * 10$ messages distributed in $N/10$ groups and $N/10$ pairs. The update consisted in $N/10$ new messages. The experiments were carried out on MySQL 5.6 running on Windows 7 in a Intel T4500 2.30GHz machine with 4GB of RAM. The database had the MySQL default indexes for primary/foreign keys.

We show our results in table 1. The title of the columns indicates the size of the database giving the number of current preexisting messages. As it can be seen, we are able able to check the integrity of the constraints in at most 3 seconds per constraint with a database state with $5{\cdot}10^6$ messages and $5{\cdot}10^4$ new insertions. In contrast, OCL2SQL/MySQL4OCL cannot afford some of the constraints after 1 hour of execution.

The result of the 3rd constraint requires to take special attention. Since adding new messages cannot cause the violation of the 3rd OCL constraint (*UserIsMemberOfOwnedGroups*), our incremental approach performs in almost constant time because the query begins from an empty auxiliary table. For the other approaches, we argue that its better performance might be because it contains one level less of subqueries in comparison to the other constraints.

5 Conclusions

We have proposed a method for efficiently checking OCL constraints by means of SQL queries that perform incrementally since only the relevant constraints and the relevant values are taken into account during the computation. In addition, they are written in such a way not to nest any other query nor procedure inside.

To achieve it, we first specify the OCL constraint violations through an EDC formalism. Such formalism offers the advantage of stating which events may cause the violation of an integrity constraint. Then, each EDC is translated into an SQL query. When doing such translation, we create some new SQL tables for temporary storing the new instances that are going to be inserted/deleted. Such auxiliary tables are used by our SQL queries to perform incrementally.

We made some experiments showing that our approach is able to check in seconds four OCL constraints over an scenario containing $5{\cdot}10^6$ instances with $5{\cdot}10^4$ updates while other approaches like OCL2SQL and MySQL4OCL could not afford some of these constraints after 1 hour of execution. As future work, we would like to extend the fragment of OCL we are dealing with and to implement in SQL the rules proposed in [16] to compare the efficiency achieved in this case.

Acknowledgements This work has been partly supported by the Ministerio de Ciencia e Innovación under project TIN2011-24747 and by the FI grant from the Secreteria d'Universitats i Recerca of the Generalitat de Catalunya. We would like also to thank the anonymous reviewers for their comments.

References

1. Olivé, A.: Conceptual Modeling of Information Systems. Springer, Berlin (2007)
2. Object Management Group (OMG): Unified Modeling Language (UML) Superstructure Specification, version 2.4.1. (2011) `http://www.omg.org/spec/UML/`.
3. Object Management Group (OMG): Object Constraint Language (UML), version 2.4. (2014) `http://www.omg.org/spec/OCL/`.
4. Hamann, L., Hofrichter, O., Gogolla, M.: On integrating structure and behavior modeling with OCL. In: International Conference on Model Driven Engineering Languages & Systems (MODELS 2012), Springer (2012) 235–251
5. Bergmann, G.: Translating OCL to graph patterns. In: Model-Driven Engineering Languages and Systems. Volume 8767 of LNCS. Springer (2014) 670–686
6. Clavel, M., Marina, E., Miguel Angel, G.d.D.: Building an efficient component for OCL evaluation. Electronic Communications of the EASST **15** (2008)
7. Demuth, B., Hussmann, H.: Using UML/OCL constraints for relational database design. In: UML99 - The Unified Modeling Language. Springer (1999) 598–613
8. Demuth, B., Wilke, C.: Model and object verification by using Dresden OCL. In: Proceedings of the Russian-German Workshop "Innovation Information Technologies: theory and practice", Russia. (2009) 81
9. Egea, M., Dania, C., Clavel, M.: MySQL4OCL: A stored procedure-based MySQL code generator for OCL. Electronic Communications of the EASST **36** (2010)
10. Oriol, X., Tort, A., Teniente, E.: Fixing up non-executable operations in UML/OCL conceptual schemas. In: Conceptual Modeling–ER 2014. (2014) 232–245
11. Uhl, A., Goldschmidt, T., Holzleitner, M.: Using an OCL impact analysis algorithm for view-based textual modelling. ECEASST **44** (2011)
12. Groher, I., Reder, A., Egyed, A.: Incremental consistency checking of dynamic constraints. In: Fundamental Approaches to Software Engineering. Springer (2010) 203–217
13. Cabot, J., Teniente, E.: Incremental integrity checking of UML/OCL conceptual schemas. Journal of Systems and Software **82**(9) (2009) 1459–1478
14. Szárnyas, G., Izsó, B., Ráth, I., Harmath, D., Bergmann, G., Varró, D.: IncQuery-D: A distributed incremental model query framework in the cloud. In: Model-Driven Engineering Languages and Systems. Volume 8767 of LNCS. Springer (2014) 653–669
15. Decker, H.: Translating advanced integrity checking technology to SQL. In: Database Integrity. (2002) 203–249
16. Miranker, D.P.: TREAT: A better match algorithm for AI production systems. In: Proc. of the 6th National Conf. on Artificial Intelligence. Volume 1 of AAAI., AAAI Press (1987) 42–47
17. Costal, D., Gómez, C., Queralt, A., Raventós, R., Teniente, E.: Improving the definition of general constraints in UML. Software & Systems Modeling **7**(4) (2008) 469–486

Validation of a security metamodel for development of cloud applications

Marcos Arjona[1], Carolina Dania[2], Marina Egea[3], Antonio Maña[1]

[1] Universidad de Málaga, Spain
[2] IMDEA Software Institute, Madrid, Spain
[3] Atos, Madrid, Spain
`marcos@lcc.uma.es, carolina.dania@imdea.org,`
`marina.egea@atos.net, amg@lcc.uma.es`

Abstract. Development of secure cloud applications requires a supportive approach that should also enable software assessment and certification by different mechanisms. These can assure by independent means that the required security is present. In this paper we present a Core Security Metamodel (CSM) that is the director of a security engineering process that also addresses security certification for cloud applications. To drive these activities with enough precision, the CSM is constrained with OCL rules that control the creation of instances of the metamodel. Due to their relevance for the security engineering process, we decided to formally check their consistency leveraging on our previous mapping from OCL to First Order Logic. We found that CVC4 returned *sat* in less than 30 seconds when we run it in finite model finder mode. Also, it automatically provided a valid CSM structural instance. Instances so obtained with CVC4 can be tuned to serve as input of the engineering process of secure cloud applications. Their automatic generation reduces the time and effort spent in the engineering process, reinforcing its supportive and practical side.

1 Introduction

Development of secure applications is a challenging task due to the evolvable risks that threaten any system under design. Even worse nowadays, the exposure of systems to cloud environments claims for a stronger development approach able to support a large number of complex security requirements and interplay in the creation of cloud applications. Most of the proposed approaches agree in the necessity to sit a solid and affordable engineering process that can prevent, from design time, non-secure states due to wrong security mechanisms used as a late solution [18]. In line with this approach, our work stems in the definition and evaluation of a security engineering process [6] for the CUMULUS [1] and the PARIS [4] EU projects. Our work proposes a complete Model Based System Engineering (MBSE) methodology to address the different stages involved in the development of secure and privacy preserving applications. It includes early stages of the architectural design that identify the security requirements. Also, it covers subsequent stages in which solutions are manipulated and system model transformations progressively applied up to the inclusion of all the security mechanisms that fulfill those security requirements.

In this paper, we are focused at the first stage of the work flow: the Core Security Metamodel (CSM), designed to gather and represent the security knowledge. The CSM and the OCL validation rules imposed on it establish a language that supports, validates and drives instance creation and subsequent steps of the engineering process. Due to their relevance for the security engineering process, we decided to formally analyze them. Thus, we mapped them to First Order Logic following our previous work [13,14,21] and then used off-the-shelf tools to run the analysis. We run Z3 [15] and CVC4 [7] as SMT solvers in the first place but they did not help us with the consistency checking. Then, we employed CVC4 as a finite model finder, which returned *sat* in less than 30 seconds and automatically provided a CSM valid structural instance. Instances so obtained are manually enhanced with security knowledge later on to serve as inputs of the engineering process of secure cloud applications. Still, their generation can reduce the time and effort, reinforcing its supportive and practical side.

Organization. In section 2 we outline related work. In section 3 we introduce the CSM, its OCL constraints and its intended use. In section 4 we summarize our previous mapping from OCL to first order logic and illustrate how CSM rules are mapped. In section 5 we report on the CVC4-based validation and instance generation. In section 6 we illustrate how instances can be enhanced to drive subsequent steps of the engineering process. Finally, in Section 7 we present conclusions and directions for future work.

2 Related work

In the software engineering arena there are a number of ready-to-use tools supporting OCL. Possibly the best starting point to get introduced to a variety of them is the OCL Portal.[4] Most of the tools that it contains ($\sim$ 11) are OCL parsers or evaluators. Also, there are 3 static verification tools, one code generator and one OCL transformation tool. For this related work, we focus in the OCL automatic verification or validation tools that could help us as alternative or complementary formal analysis means to the ones that we have already applied to CSM helped by Z3 [15] and CVC4 [7]. [5]

Recently published, the systematic review [17] deeply reports on 18 research lines on static verification of UML-like structural diagrams. Taking these research results as the starting point, we decided to focus here only on those for which the tool associated is ready to download or use from a website and supports automatic analysis for a significant subset of OCL. We consider these criteria as essential criteria for analysis tools to be actually of use in real development processes.

Alternative tools to CVC4 for our goal could be, in principle, the ones reported next. UMLtoCSP [12] and EMFtoCSP [16] both provide bounded automatic verification of UML (resp. EMF) models annotated with OCL constraints. The users must limit the search space by explicitly indicating the number of objects in each class, the number of links of each association and the possible values of each attribute. When the tool cannot find a satisfying instance within the specified search space, this does not mean that the property does not hold, because it can still hold for instances outside that search

---

[4] `http://www-st.inf.tu-dresden.de/oclportal`, last visited in July 2014.

[5] We note that the use of these tools was eased by the output of our tool [21] that maps OCL to FOL being SMT-LIB [8] standard.

space (and the user may try to verify the property with wider intervals). In the same vein, the tool UML2Alloy [5] performs bounded verification in relational logic. Also, the extension of USE tool with relational logic for satisfiability checking [19]. From this tool we much appreciate as an usability advantage the facility of graphical display of the instances found as object diagrams. Finally, similar analysis can be performed using propositional logic [20], but we could not find the BV-SAT available from a web page, although it is reported as an automatic tool in [17]. Yet, there is a major advantage in our approach thanks to our mapping [14], the use we make of SMT solvers supports the OCL 4-valued logic, which is not supported by none of the tools described before.

Although they do not perform fully automatic analysis, we consider complementary tools interactive theorem provers like, e.g. HOL-OCL [10] or the Key tool [9]. The fact that they are not fully automatic impact their use that requires too high mathematical background, precluding them from standard software development practice. As a final remark, we note that HOL-OCL supports the 4-valued logic of OCL.

3 The Core Security Metamodel

In this section we explain the Core Security Metamodel (CSM) which allows the description of the security related knowledge that needs to be considered in the development of secure cloud applications. Reflecting the complexity of the security field, the CSM is a composition of 6 sub-models that address different security expertise subareas. Thus, CSM instantiation is facilitated by these groups of related elements which are displayed in the metamodel with different colors, as shown in Figure 1.[6] Next we describe these sub-models, also to understand how they fit together.

Requirement sub-model (green): it is used to qualify security and certification requirements by means of security valuators, mechanisms and certified services.

Property sub-model (yellow): it is used to describe abstract security properties involved in a security requirement, specifying its attributes and values.

Domain sub-model (brown): it is used to describe the domain or context of the CSM instance, identifying the assets to be protected.

Solution sub-model (pink): it is used to show how the security requirements will be achieved by means of solutions and security mechanisms.

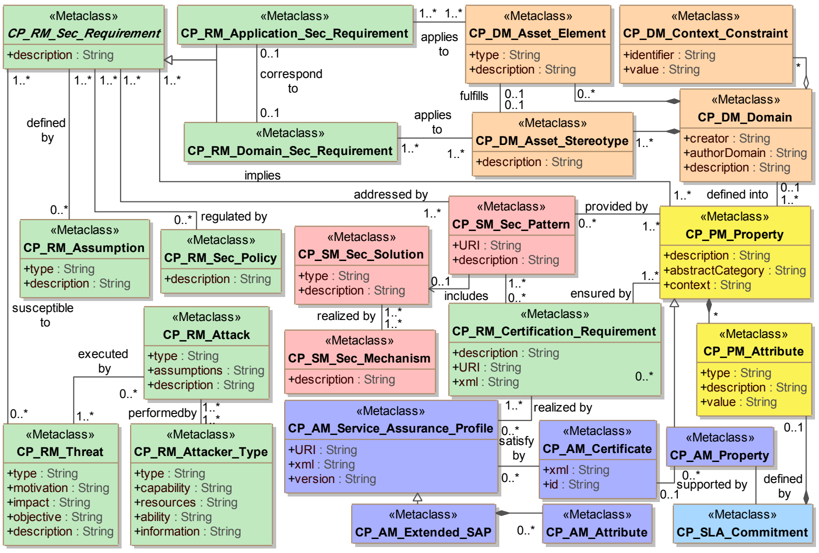
Assurance sub-model (blue): it is used to specify the assurance profile and the certification-related elements that would fulfill the certification requirements.

Service Level Agreement sub-model (light blue): it is used to specify SLA agreements that may affect the security properties.

The CUMULUS engineering process aims not only at supporting experts to express their expertise into a model, but also to orchestrate an automated sorting and processing of that information to make it accessible and useful for non security experts. The effectiveness of this approach heavily relies on the OCL validation system which supports three goals in the CSM instantiation activity:

---

[6] CSM has been already proved its use for real applications to integrate security mechanisms in high risk environments [23,22], but using a different security engineering process in the context of the SecFutur Project (`http://www.secfutur.eu`).



«Metaclass» CP_RM_Sec_Requirement +description : String
«Metaclass» CP_RM_Application_Sec_Requirement
«Metaclass» CP_DM_Asset_Element +type : String +description : String
«Metaclass» CP_DM_Context_Constraint +identifier : String +value : String
«Metaclass» CP_DM_Domain +creator : String +authorDomain : String +description : String
«Metaclass» CP_RM_Domain_Sec_Requirement
«Metaclass» CP_DM_Asset_Stereotype +description : String
«Metaclass» CP_RM_Assumption +type : String +description : String
«Metaclass» CP_RM_Sec_Policy +description : String
«Metaclass» CP_SM_Sec_Solution +type : String +description : String
«Metaclass» CP_SM_Sec_Pattern +URI : String +description : String
«Metaclass» CP_PM_Property +description : String +abstractCategory : String +context : String
«Metaclass» CP_RM_Attack +type : String +assumptions : String +description : String
«Metaclass» CP_SM_Sec_Mechanism +description : String
«Metaclass» CP_RM_Certification_Requirement +description : String +URI : String +xml : String
«Metaclass» CP_PM_Attribute +type : String +description : String +value : String
«Metaclass» CP_RM_Threat +type : String +motivation : String +impact : String +objective : String +description : String
«Metaclass» CP_RM_Attacker_Type +type : String +capability : String +resources : String +ability : String +information : String
«Metaclass» CP_AM_Service_Assurance_Profile +URI : String +xml : String +version : String
«Metaclass» CP_AM_Certificate +xml : String +id : String
«Metaclass» CP_AM_Property
«Metaclass» CP_AM_Extended_SAP
«Metaclass» CP_AM_Attribute
«Metaclass» CP_SLA_Commitment
applies to
correspond to
fulfills
applies to
implies
defined by
addressed by
regulated by
provided by
defined into
includes
ensured by
realized by
susceptible to
executed by
performedby
realized by
satisfy by
supported by
defined by
1..* 0..1 0..* 1 *

Fig. 1. Core Security Metamodel

1. *Perform an active validation of the modeling process.* This validation raises a warning if the instance does not conform to the metamodel. It also highlights the pieces of information that are missing or wrong. This validation helps experts to avoid wrong specifications that would impact the run time of the system.
2. *Check that required information is present.* It validates whether a valid CSM instance lacks information that is needed by the engineering activities. E.g., transitive association between specific components, empty attributes, etc..
3. *Guide experts during the creation of the CSM instance.* They are guided towards the next piece of information that is needed and its goal in the engineering process.

Therefore the list of OCL constraints is expected to be consistent and reactive enough to support constant interaction with it. Our rules drive an incremental validation system that is gradually triggered within the MagicDraw modelling framework [3].

OCL Constraints. The OCL validation package is composed of 33 rules. Out of these, 27 are structural constraints restraining metamodel associations. Next, we introduce those OCL constraints that do not deal directly with multiplicities.

1. A domain instance must exist and be unique

   inv: `CP_DM_Domain.allInstances()->size() = 1`

2. A certification requirement needs to be associated with a service assurance profile.

   context: `CP_RM_Certification_Requirement` **inv:**
   `(not self.URI.oclIsUndefined()) implies`
   `self.service_assurance_profile->notEmpty()`

3. A certification requirement must be linked directly and through a security pattern to a security requirement and a property

context: `CP_PM_Property` **inv:** `self.certification_requirement->notEmpty()`
`implies self.certification_requirement.sec_pattern.sec_requirement`
`->intersection(self.sec_requirement)->notEmpty()`

4. A certification requirement should be directly linked to a property and a security pattern for that property

   context: `CP_RM_Certification_Requirement` **inv:**
   `self.property->intersection(self.sec_pattern.property)->notEmpty()`

5. An asset stereotype is set up over an asset element that must be considered by an application security requirement of that asset stereotype domain

   context: `CP_DM_Asset_Stereotype` **inv:** `(not`
   `self.asset_element.oclIsUndefined()) implies`
   `self.domain_sec_requirement.application_sec_requirement.`
   `asset_element->includes(self.asset_element)`

6. A security pattern must display a security solution

   context: `CP_SM_Sec_Pattern` **inv:** `(not self.URI.oclIsUndefined())`
   `implies (not self.sec_solution.oclIsUndefined())`

4 Using OCL2FOL to map CSM into First Order Logic

In this section, we first recall our mapping from metamodels and OCL constraints to FOL [13,14,11]. Then, we map the most illustrative constraints introduced in section 3.

- *Type-predicates:* Metamodels' classes are mapped to unary boolean functions. E.g., the class `CP_SM_Sec_Solution` is mapped to $\mathrm{CPSMSecSolution} : \mathrm{Int} \rightarrow \mathrm{Bool}$;
- There are two predicates $\mathrm{isNull} : \mathrm{Int} \rightarrow \mathrm{Bool}$ and $\mathrm{isInvalid} : \mathrm{Int} \rightarrow \mathrm{Bool}$, which return `true` to represent the values `null` or `invalid` (resp.);
- Objects variables are mapped to integer variables, e.g, an object variable cl of type `CP_SM_Sec_Solution` is mapped to an integer variable cl, such that $\mathrm{CPSMSecSolution}(cl)$ holds;
- *Attribute-functions:* Attributes are mapped to integer functions, e.g., the attribute `id` of the class `CP_AM_Certificate` is mapped to a function $\mathrm{CPAMCid} : \mathrm{Int} \rightarrow \mathrm{Int}$.[7]
- *Association-predicates:* Association-ends are mapped, according to their multiplicity, either to predicates or functions. E.g., the association `realizedby` between `CP_AM_Service_Assurance_Profile` and `CP_RM_Certification_Requirement` is mapped into $\mathrm{CPAMSAPrealizedby} : \mathrm{Int} \times \mathrm{Int} \rightarrow \mathrm{Bool}$.
- For each pair of different classes, e.g. `CP_SM_Sec_Solution` and `CP_DM_Sec_Mechanism` (that are not sub-classes of any other class), the predicates $\mathrm{CPSMSecSolution}$ and $\mathrm{CPDMSecMechanism}$ must be disjoint, i.e: $\forall(x)\ \neg(\mathrm{CPSMSecSolution}(x) \wedge \mathrm{CPDMSecMechanism}(x))$. Similar formulas are included for all type-predicates.
- Also, we map inheritance relations. E.g., in Figure 1, there is an inheritance relation from the parent class `CP_RM_Sec_Requirement` to the children classes `CP_RM_Application_Sec_Requirement` and `CP_RM_Domain_Sec_Requirement`.

---

[7] For the sake of simplicity, we do not consider attributes with type object, neither multivalued attributes. Also, boolean attributes are always mapped into an integer attribute.

We map this relation as follows: $\forall(x)(\text{CPRMApp}-\text{SecReq}(x) \Rightarrow \text{CPRMSecReq}(x))$ and $\forall(x)(\text{CPRMDomainSecReq}(x) \Rightarrow \text{CPRMSecReq}(x))$.
Since, `CP_RM_SecRequirement` is an abstract superclass, then the following assertion are included: $\forall(x)\ \neg(\text{CPRMAppSecReq}(x) \wedge \text{CPRMDomainSecReq}(x))$ and $\forall(x)(\text{CPRMSecReq}(x) \Rightarrow (\text{CPRMAppSecReq}(x) \vee \text{CPRMDomainSecReq}(x)))$.

- `OCL Boolean`-expressions are translated to formulas, which essentially mirror the logical structure of the OCL expressions, e.g., for the operations `or`, `and`, `implies`, `not`, `notEmpty`, `includes`, `oclIsUndefined`, `forAll`, `exists`, $=$, $\neq$; e.g., `CP_PM_Pro- perty.allInstances()->notEmpty()` is mapped into: $\exists(x)(\text{CPPMProperty}(x))$.
- `OCL Integer`-expressions are basically copied, e.g. $+$, $-$, $*$. Currently, we only cover simple operations (i.e., $=$ and $<>$) over `OCL String`-expressions.
- `OCL Collection`-expressions are translated to *fresh* predicates that augment the signature of the specification. Their meaning is defined by additional formulas also generated by the mapping. E.g., `select`, `collect`, `intersection`, etc..

Next we show the mapping of the CSM constraints numbered 2 and 4 in section 3. [8] Their mapping well represents the one required for the other constraints. Note that, for the constraint 4 two new *fresh* predicates are created: Collect1 and Intersection1.

[2]. `CP_RM_Certification_Requirement.allInstances()->forAll(c|not(c.URI.oclIsUndefined()) implies (not s.service_assurance_profile->notEmpty()))`

$$\forall(x)(\text{CPRMCertificationRequirement}(x) \wedge \neg(\text{isNull}(\text{CPRMCRurl}(x)) \vee \text{isInvalid}(x)) \Rightarrow \exists(y)(\text{CPAMServiceAssuranceProfile}(y) \wedge \text{CPRMCRrealizedby}(y,x)))$$

[4]. `CP_RM_Certification_Requirement.allInstances()->forAll(c|c.property->intersection(c.sec_pattern->collect(p|p.property))->notEmpty())`

$$\forall(x)(\text{CPRMCertificationRequirement}(x) \Rightarrow \exists(y)(\text{Intersection1}(\text{x},\text{y})))$$
$$\forall(x,y)(\text{Intersection1}(x,y) \Leftrightarrow (\text{CPPMPensuredBy}(y,x) \wedge \text{Collect1}(x,y)))$$
$$\forall(x,y)(\text{Collect1}(x,y) \Leftrightarrow \exists(z)(\text{CPSMSecPattern}(z) \wedge \text{CPSMSPcertification}(z,x) \wedge \text{CPPMPprovidedBy}(y,z)))$$

5 Core Security metamodel validation and instance generation

In this section we explain the analysis that we perform on the OCL constrained CSM metamodel once it is translated to FOL.[9] We first tried to check whether the OCL constraints imposed on the CSM were or not unsatisfiable (and generate an example in the latter case) by feeding them to the SMT solvers Z3 [15] and CVC4 [7]. However, after more than 3 hours running, they did not return any result, and we decided to stop them. We know that this lack of result from Z3 and CVC4 is due to the fact that current techniques for dealing with quantified formulas in SMT are generally incomplete. In particular, they usually have problems to prove the unsatisfiability of a formula with

---

[8] We want to note that our mapping is not yet complete but it does cover a sufficiently significant subset of the OCL language.

[9] Translation available at http://www.software.imdea.org/~dania/tools/csm.html.

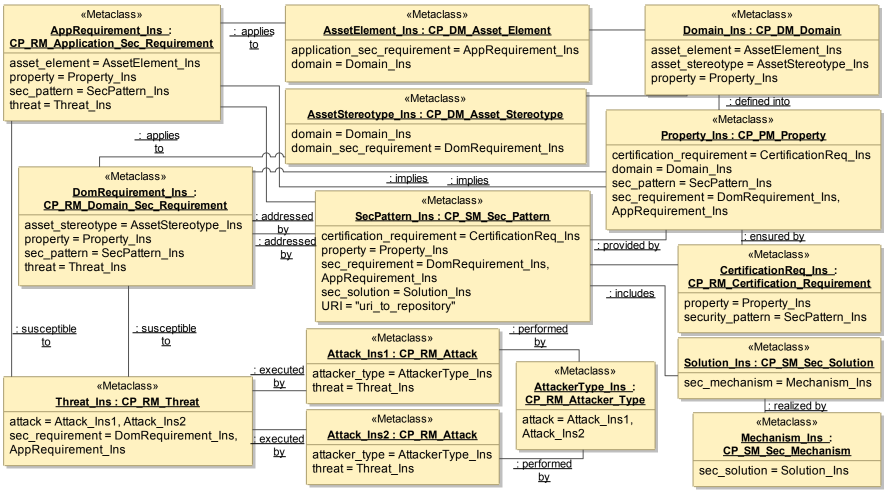

«Metaclass»
AppRequirement_Ins : CP_RM_Application_Sec_Requirement
asset_element = AssetElement_Ins
property = Property_Ins
sec_pattern = SecPattern_Ins
threat = Threat_Ins
: applies to
«Metaclass»
AssetElement_Ins : CP_DM_Asset_Element
application_sec_requirement = AppRequirement_Ins
domain = Domain_Ins
«Metaclass»
Domain_Ins : CP_DM_Domain
asset_element = AssetElement_Ins
asset_stereotype = AssetStereotype_Ins
property = Property_Ins
«Metaclass»
AssetStereotype_Ins : CP_DM_Asset_Stereotype
domain = Domain_Ins
domain_sec_requirement = DomRequirement_Ins
: defined into
«Metaclass»
Property_Ins : CP_PM_Property
certification_requirement = CertificationReq_Ins
domain = Domain_Ins
sec_pattern = SecPattern_Ins
sec_requirement = DomRequirement_Ins, AppRequirement_Ins
: applies to
: implies
: implies
«Metaclass»
DomRequirement_Ins : CP_RM_Domain_Sec_Requirement
asset_stereotype = AssetStereotype_Ins
property = Property_Ins
sec_pattern = SecPattern_Ins
threat = Threat_Ins
: addressed by
: addressed by
«Metaclass»
SecPattern_Ins : CP_SM_Sec_Pattern
certification_requirement = CertificationReq_Ins
property = Property_Ins
sec_requirement = DomRequirement_Ins, AppRequirement_Ins
sec_solution = Solution_Ins
URI = "uri_to_repository"
: provided by
: ensured by
«Metaclass»
CertificationReq_Ins : CP_RM_Certification_Requirement
property = Property_Ins
security_pattern = SecPattern_Ins
: includes
«Metaclass»
Solution_Ins : CP_SM_Sec_Solution
sec_mechanism = Mechanism_Ins
: realized by
«Metaclass»
Mechanism_Ins : CP_SM_Sec_Mechanism
sec_solution = Solution_Ins
: susceptible to
: susceptible to
: performed by
«Metaclass»
Attack_Ins1 : CP_RM_Attack
attacker_type = AttackerType_Ins
threat = Threat_Ins
: executed by
«Metaclass»
AttackerType_Ins : CP_RM_Attacker_Type
attack = Attack_Ins1, Attack_Ins2
«Metaclass»
Threat_Ins : CP_RM_Threat
attack = Attack_Ins1, Attack_Ins2
sec_requirement = DomRequirement_Ins, AppRequirement_Ins
: executed by
«Metaclass»
Attack_Ins2 : CP_RM_Attack
attacker_type = AttackerType_Ins
threat = Threat_Ins
: performed by

Fig. 2. Automatically generated instance of the security metamodel presented in the Figure 1.

universal quantifiers (our specification is plenty of them).[10] Then, we decided to employ CVC4 as a finite model finder on our specification to check its satisfiability because the input required by it is the same input for the SMT solvers. CVC4 performed a bounded checking and succeeded by returning *sat* and automatically producing finite instances that conform to the OCL constrained CSM. Let us note that to work with the finite model finder CVC4, since the output of our tool [21] is SMT-LIB, we only needed to change in our mapping the sorts Int by a finite sort U. CVC4 run less than 30 seconds to answer SAT and return a simple CSM instance.

Then, we included additional OCL constraints to require a defined URI for all instances of `CP_SM_Sec_Pattern`, to contain a minimum of two `CP_RM_Attack` instances, and at least one instance of each of the following classes: `CP_RM_Attack_Type`, `CP_RM_Certification_Requirement`, `CP_SM_Sec_Solution` and `CP_SM_Sec_Mechanism`. They ensure that generated instances contain at least a minimum amount of information that makes them meaningful for a security expert. Then, we run CVC4 again with these additional constraints, and after less than 1 minute, the instance that we depict in Figure 2 was returned. The instances so obtained with CVC4 match structurally those obtained following the security engineering process and would allow to skip some of its steps (provided that we could automatically tailor the instances obtained by CVC4 to serve as inputs for the modeling framework). As we show next, these instances can be enhanced with knowledge (semantics) from the security domain so as they can serve as input for subsequent steps of the security engineering process.

6 Security enhanced CSM instances

As we already mentioned, the CSM is part of an assisted methodology, supported by the CUMULUS modelling tool [2], that has been initially conceived to take advantage of

---

[10] More specifically, in this case the problem is introduced by how we map certain types of association ends into FOL. If we do not include their translation, the SMT solvers terminate, but the instances they return are not always valid instances.

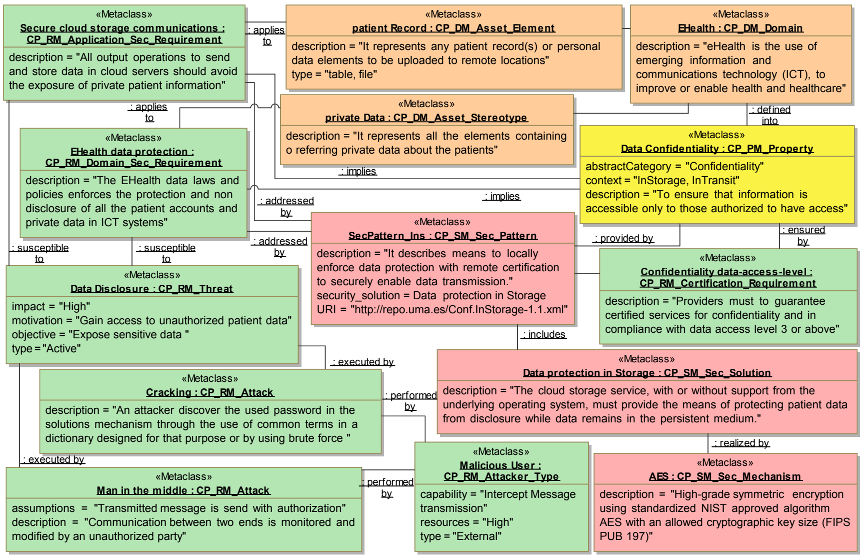


Fig. 3. Domain Security Metamodel

the multiple capabilities provided by the MagicDraw framework [3], particularly of its OCL validation engine. This methodology aims at supporting security experts to specify and communicate to system engineers how to solve security issues for cloud applications. When security experts design their models, i.e., CSM instances, the CUMULUS framework guides the construction of these instances (Domain Security Metamodels-DSMs) with the OCL rules that are continuously validated over them, raising warnings that claim for mandatory elements that are not yet present or errors. This process establishes a common format for the knowledge modeled, ensuring its applicability later on. The resulting instance (i.e., a DSM) is a validated artifact ready to transform security requirements into certification requirements and links to the solutions and mechanisms able to assure local system architectures and their interaction with cloud platforms [6].

For example, the rule [1] in section 3 requires a unique domain instance. Experts dealing with security knowledge in the *EHealth* domain in cloud environments may describe a model for non security experts so as to improve a health care process (we follow Fig. 3). Domain specification is critical to upload DSMs into the appropriate repository, to classify the DSM content adequately. Once a valid domain instance has been created, the validation system triggers those rules that are not yet satisfied so as the model has to be extended to fulfill them. In our example, the framework requests at least one `Property` and one `Asset_Stereotype` instance to be linked with the `Domain`, stemming from the CSM multiplicities and the constraint descriptions. Our DSM is extended with *private data* as an asset stereotype to *represent all the elements containing private patient data* and the security property *Data Confidentiality* (that would *ensure that information is accessible only to authorized users*). This modus operandi is repeated until DSM fully conforms structurally to the CSM and its OCL constraints.

For the sake of space, we do not describe here in full the DSM creation process. But we further describe the DSM instance in Fig. 3. It contains as security requirements

EHealth data protection and *Secure cloud storage communications*, both associated to the threat *Data Disclosure*. In addition, we have created an additional asset *patient record*, potential attacks as *Cracking* or *Man in the middle* and, finally, a common attacker type *Malicious User*. Probably, the most important part of a DSM is the selection of security patterns and certification requirements. The issue to be solved is described in the pattern, in our example, *means to locally enforce data protection with remote certification to securely enable data transmission*. How it should be guaranteed is specified by the certification requirement, in our example, *the usage of certified services for confidentiality and in compliance with data access level 3 or above*. Both plain descriptions have consequences in the security engineering process because they limit the solutions to be deployed for cloud applications. Recalling subsection 3, the last constraint requires that for a security pattern and a solution to be linked, the URI attribute of the pattern must be defined. This constraint demands intervention of the security expert since they search and select from existing repositories, through an API provided by the framework, a suitable pattern that also links a target solution, e.g., *Data protection in Storage* and a security mechanism, e.g., *AES*. As a result of the modelling process, security experts provide a complete artifact ready to fulfill security requirements addressing both the local mechanisms and the remote certification requirements.

Finally, we remark that both instances shown in Figures 2 and 3 resp., are structurally identical. Thus, the engineering process receive a shortcut from the use of automatic finite model finders that ease the path and reduce the time required to build instances since they can automatically generate them. Then, instances can be enhanced with security domain specific knowledge and trigger subsequent engineering activities.

7 Conclusions and Future Work

In this paper we have introduced a security metamodel (CSM) that is constrained by 33 OCL rules that drive the engineering of secure cloud applications. We formally analyzed this metamodel, that is both complex and large, and its constraints, to gain confidence on their consistency and adequacy for the engineering process. We used our previous work, OCL2FOL [13,14,21] to automatically map the metamodel and its constraints to first order logic. Then, we employed successfully a finite model finder, CVC4, that returns '*sat*' for the resulting specification. We also illustrated how the instances automatically generated by CVC4 conform to the CSM and its constraints, and are enhanced with domain security knowledge to get ready to trigger the remaining engineering activities. The automated approach generates an instance that matches one obtained following the engineering process. Based on these results we can say that our formal analysis besides providing higher assurance of the adequacy of the CSM and its rules, also reduces the time and effort required from the security experts in the initial stage of the CUMULUS engineering process. Particularly, since the automatic valid instances generation. Yet, we will need to implement a converter from the CVC4 instances to a valid model input format for MagicDraw to automate the process based on instance generation.

Acknowledgement. This research was partially supported by the 7th EU Framework Programme project CUMULUS (Certification infrastructure for multi-layer cloud services) grant no. 318580 and PARIS (Privacy Preserving Infrastructure for Surveillance) grant no. 312504, and by the Spanish Ministry of Economy and Competitiveness Project "StrongSoft" (TIN2012-39391-C04-04).

References

1. CUMULUS Project. `http://cumulus-project.eu/`.
2. D4.2: Tools supporting CUMULUS-aware engineering process v1. `http://cumulus-project.eu/index.php/public-deliverables`.
3. MagicDraw Modelling Tool. `http://www.nomagic.com/products/magicdraw.html`.
4. PARIS Project. `http://www.paris-project.org/`.
5. K. Anastasakis, B. Bordbar, G. Georg, and I.Ray. UML2Alloy: A Challenging Model Transformation. In *MoDELS 2007*, volume 4735 of *LNCS*. Springer, 2007. Tool available at `http://www.cs.bham.ac.uk/~bxb/UML2Alloy/download.php`, last access: June 2014.
6. M. Arjona, R. Harjani, A. Muñoz, and A. Maña. An Engineering Process to Address Security Challenges in Cloud Computing, 3rd ASE International Conference on Cyber Security, 2014.
7. C. Barrett, C. L. Conway, M. Deters, L. Hadarean, D. Jovanovic, T. King, A. Reynolds, and C. Tinelli. CVC4. pages 171–177, 2011.
8. C. Barrett, A. Stump, and C. Tinelli. The SMT-LIB Standard: Version 2.0. In *Proc. of the 8th International Workshop on Satisfiability Modulo Theories*, 2010.
9. B. Beckert, U. Keller, and P. H. Schmitt. Translating OCL into First-order Predicate Logic. In *In Proc. of VERIFY Workshop at Federated Logic Conferences (FLoC)*, 2002.
10. A. D. Brucker and B. Wolff. HOL-OCL: A Formal Proof Environment for UML/OCL. In *FASE 2008*, volume 4961 of *LNCS*. Springer, 2008.
11. F. Büttner, M. Egea, and J. Cabot. On Verifying ATL Transformations Using 'off-the-shelf' SMT Solvers. In *MoDELS*, volume 7590 of *LNCS*, pages 432–448. Springer, 2012.
12. J. Cabot, R. Clarisó, and D. Riera. UMLtoCSP: a tool for the formal verification of UML/OCL models using constraint programming. In *ASE 2007, Proc.* ACM, 2007. Tool available at `http:/gres.uoc.edu/UMLtoCSP/`.
13. M. Clavel, M. Egea, and M. A. García de Dios. Checking Unsatisfiability for OCL Constraints. *Electronic Communications of the EASST*, 24:1–13, 2009.
14. C. Dania and M. Clavel. OCL2FOL+: Coping with Undefinedness. In *Proc. of the MODELS 2013 OCL Workshop*, volume 1092 of *CEUR Workshop Proceedings*, pages 53–62, 2013.
15. L. Mendonça de Moura and N. Bjørner. Z3: An Efficient SMT Solver. In C. R. Ramakrishnan and J. Rehof, editors, *TACAS*, volume 4963 of *LNCS*, pages 337–340. Springer, 2008.
16. C. A. González, F. Büttner, and Jordi Cabot. EMFtoCSP: A Tool for the Lightweight Verification of EMF Models. In *FormSERA*, pages 44–50, 2012. Tool at `https://code.google.com/a/eclipselabs.org/p/emftocsp/`.
17. C. A. González and J. Cabot. Formal verification of static software models in MDE: A systematic review. *Information & Software Technology*, 56(8):821–838, 2014.
18. R. Harjani, M. Arjona, A. Muñoz, and A. Maña. Towards an Engineering Process for Certified Multilayer Cloud Services, Layered Assurance Workshop. ASAC. 2013.
19. M. Kuhlmann, L. Hamann, and M. Gogolla. Extensive Validation of OCL Models by Integrating SAT Solving into USE. In *TOOLS 2011*, volume 6705 of *LNCS*, pages 290–306. Springer, 2011. Tool available at `http://sourceforge.net/projects/useocl/`.
20. M.Soeken, R.Wille, and R.Drechsler. Encoding OCL Data Types for SAT-Based Verification of UML/OCL Models. In *TAP*, volume 6706 of *LNCS*, pages 152–170. Springer, 2011.
21. OCL2FOL Project, 2012. `http://www.actiongui.org`, see OCL2FOL and OCL2FOL+.
22. J. F. Ruiz, A. Maña, M. Arjona, and J. Paatero. Emergency Systems Modelling using a Security Engineering Process. In *Proc. of 3rd Int. Conf. SIMULTECH*. SciTePress, 2013.
23. J.F. Ruiz, A. Rein, M. Arjona, A. Maña, A. Monsifrot, and M. Morvan. Security Engineering and Modelling of Set-Top Boxes. In *Proc. of ASE/IEEE BioMedCom*, 2012.

Towards a Tool for Featherweight OCL: A Case Study On Semantic Reflection

Delphine Longuet[1], Frédéric Tuong[2] and Burkhart Wolff[1]

[1] Univ Paris-Sud, LRI UMR8623, Orsay, F-91405
CNRS, Orsay, F-91405
`{delphine.longuet, burkhart.wolff}@lri.fr`
[2] Univ Paris-Sud, IRT SystemX, 8 av. de la Vauve, Palaiseau, F-91120
`frederic.tuong@{u-psud, irt-systemx}.fr`

Abstract We show how modern proof environments comprising code generators and reflection facilities can be used for the effective construction of a *tool* for OCL. For this end, we define a UML/OCL meta-model in HOL, a meta-model for Isabelle/HOL in HOL, and a compiling function between them over the vocabulary of the libraries provided by Featherweight OCL. We use the code generator of Isabelle to generate executable code for the compiler, which is bound to a USE tool-like syntax integrated in Isabelle/Featherweight OCL. It generates for an arbitrary class model an object-oriented datatype theory and proves the relevant properties for casts, type-tests, constructors and selectors automatically.
Keywords: UML, OCL, Formal Semantics, Isabelle/HOL, Reflection.

1 Introduction

In this paper, we present a radically different approach to design, model, and implement a tool for UML and OCL. In the conventional approach, developers use hand-written Java programs complemented by the components generated by parsing generators [8] and by UML frameworks like OMG's MetaObject Facility (MOF)[10] or Eclipse Modeling Framework Project (EMF)[9]. In contrast, in our approach, we use the Isabelle Framework to generate a tool derived from a formal semantics for (some part of) UML/OCL based on earlier work for Featherweight OCL [3, 4, 6]. While the resulting tool — including user interface, document generator, code generator, and proof support — cannot compete to direct implementations such as [1, 12] with respect to speed of code-generation and completeness of the supported OCL language, our front-end has its value as a semantically founded reference implementation.

As a formalized theory in Isabelle/HOL, Featherweight OCL provides:

- the *algebraic layer* that contains the definitions of the four-valued logics (including `invalid` and `null`) with two equalities as well as theories for Integers and Sets,
- the *state layer* describing state-related operations like `allInstances()`, and
- the *object-oriented data-type layers* giving semantics to UML class models over this, comprising the theory of accessors, type casts and tests.

In this paper, we target to mechanize the latter. This means that our tool generates proven theorems that make the properties of object-oriented data-type theories explicit and which are needed as background-theory for other tools based on automated deduction striving for semantic compliance with the standard. As input, our tools takes a *class model*, which comprises:

1. Classes and class names (written as C_1, …, C_n), which become types of data in OCL. Class names declare two projector functions to the set of all objects in a state: C_i`.allInstances()` and C_i`.allInstances@pre()`,
2. an inheritance relation $\_ < \_$ on classes and a collection of attributes A associated to classes,
3. two families of accessors for each attribute a and object X in a class definition (denoted by X`.`a $:: C_i \to A$ and X`.`a`@pre` $:: C_i \to A$ for $A \in \{\text{Boolean}, \text{Integer}, \dots, C_1, \dots, C_n\}$),
4. two families of operation declarations f_{op} for each class,
5. type casts that can change the static type of an object of a class (denoted by X`.oclAsType(`C_i`)` of type $C_j \to C_i$)
6. two dynamic type tests (denoted by X`.oclIsTypeOf(`C_i`)` and X`.oclIsKindOf(`C_i`)`),
7. and last but not least, for each class name C_i, an instance of the overloaded referential equality (written $\_ \doteq \_$).

We use the notation $e :: \tau$ to say that some expression e has the static type τ. Note that the phenomenon of dynamic types ("the type of an object at creation time") vs. static types ("the type inferred by the type inference in presence of possibly implicit casts") is characteristic for statically typed object-oriented languages such as Java or C++; apart from syntax, the object-oriented data model presented here is in no way specific to UML/OCL. Finally, for each function induced by the class model rules must be derived treating strictness, null, definedness, etc… Moreover, the subtype and inheritance relationship between objects must be expressed, for example by the rule:

$$(X :: C_k)\texttt{.oclAsType(}C_j\texttt{).oclAsType(}C_k\texttt{)} = X$$

where X is an object and C_k is a sub-type of C_j (i. e. $C_k < C_j$). This rule means that objects can be losslessly cast up and down again; this property is the key for the implementation of generic classes in Java and should also hold in UML.

While in [4] we described the *construction* of object-oriented data-types for UML class models *conceptually* and demonstrated it by an example containing definitions and rules proven by hand, we go in the present work one step further: We effectively *construct* an Isabelle plug-in for UML class models, that parses them in concrete input syntax (inspired by the USE tool) and compiles them to the necessary definitions, proofs, and infrastructure for execution and animation. As a by-product, UML class models can be directly edited *inside* Isabelle theory files, thus inheriting the Isabelle infrastructure including IDE, code generators, document generators and — last but not least — the proof environment for modeling and reasoning.

2 Background: Isabelle and UML/OCL

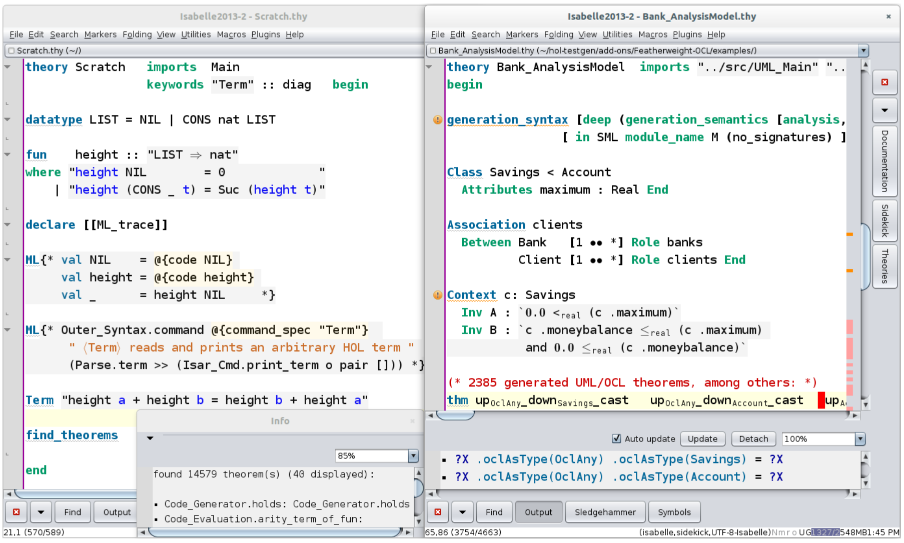
2.1 Isabelle: A Guided Tour through the Framework

In this paper, we want to emphasize the use of Isabelle as a generic **technical** framework. As such, it offers the possibility to "drive" the core-engine by user-programmed Standard ML (SML) programs in a logically safe way; a system configuration ("session") can thus contain logical definitions, proofs, text-documentation as well as code for tactic support as well as system extensions.

In order to demonstrate some relevant system features, we present a screenshot in Fig. 1. The left window shows a session based on Isabelle/HOL that consists of the only file `Scratch.thy`. One recognizes the header including theory and the "`imports` *Main*" clause ("*Main*" is a synonym for Isabelle/HOL) and then a sequence of subsections — called *commands* — introduced by a blue keyword: datatype, fun, declare, ML, find_theorems, thm... User-interaction to Isabelle is *document oriented*, i. e. each file belonging to a session is annotated by the prover while editing it as usual by modern IDEs. This annotation can consist, for example, in:

- colors (here: the underlying white indicates that Isabelle checked these commands and executed them without error),
- types (to be explored by tool-tips via the hovering-gesture),
- or values associated to computations inside these commands (displayed in the "Output" window when pointing to them; see Fig. 1 (right-below)).

Isabelle sessions can be extended by *user-defined commands*, a feature we use for defining Term, or for our own textual class model syntax in Fig. 1 (right):


Isabelle2013-2 - Scratch.thy
File Edit Search Markers Folding View Utilities Macros Plugins Help
☐ Scratch.thy (~/)

```
theory Scratch   imports  Main
                 keywords "Term" :: diag   begin

datatype LIST = NIL | CONS nat LIST

fun    height :: "LIST ⇒ nat"
where "height NIL        = 0              "
    | "height (CONS _ t) = Suc (height t)"

declare [[ML_trace]]

ML{* val NIL    = @{code NIL}
     val height = @{code height}
     val _      = height NIL    *}

ML{* Outer_Syntax.command @{command_spec "Term"}
       " ⟨Term⟩ reads and prints an arbitrary HOL term "
       (Parse.term >> (Isar_Cmd.print_term o pair [])) *}

Term "height a + height b = height b + height a"

find_theorems

end
```

```
found 14579 theorem(s) (40 displayed):

▪ Code_Generator.holds: Code_Generator.holds
▪ Code_Evaluation.arity_term_of_fun:
```

☒
Find
Output
21,1 (570/589)
Isabelle2013-2 - Bank_AnalysisModel.thy
File Edit Search Markers Folding View Utilities Macros Plugins Help
☐ Bank_AnalysisModel.thy (~/hol-testgen/add-ons/Featherweight-OCL/examples/)
Documentation
Sidekick
Theories

```
theory Bank_AnalysisModel  imports "../src/UML_Main" "..
begin

generation_syntax [deep (generation_semantics [analysis,
                  [ in SML module_name M (no_signatures) ]

Class Savings < Account
  Attributes maximum : Real End

Association clients
  Between Bank   [1 •• *] Role banks
          Client [1 •• *] Role clients End

Context c: Savings
  Inv A : `0.0 <real (c .maximum)`
  Inv B : `c .moneybalance ≤real (c .maximum)
          and 0.0 ≤real (c .moneybalance)`

(* 2385 generated UML/OCL theorems, among others: *)
thm upOclAny_downSavings_cast   upOclAny_downAccount_cast   upA
```

```
▪ ?X .oclAsType(OclAny) .oclAsType(Savings) = ?X
▪ ?X .oclAsType(OclAny) .oclAsType(Account) = ?X
```

Fig. 1. Screenshot of Isabelle/jEdit: Standard HOL (left) and UML/OCL (right)

In the following, we describe the effect of commands in more detail (an in-depth treatment can be found in the implementation manual); the resulting class model compiler will be an application of the techniques demonstrated here.

The datatype command. The easiest way to understand this command is to view it as a kind of macro (albeit its syntax is inspired by functional programming languages) for the type declaration of `LIST` and a number of constant definitions and theorems. Some of these definitions construct a model of the constructors and derive its properties such as $\texttt{NIL} \neq \texttt{CONS}\ n\ mm$, $\texttt{CONS}\ n\ mm = \texttt{CONS}\ n'\ mm' \to n = n'$ and the induction rule for the type `LIST`. Thus, this command constructs the *theory* of a freely generated data-type.

The fun command. Similarly, the fun command allows for the declaration of recursive functions with pattern-matching. Again a conservative construction using a recursor is used; the derivation of the equations $\texttt{height NIL} = 0$ and $\texttt{height}(\texttt{CONS}\ n\ m) = \texttt{Suc}(\texttt{height}\ m)$ is done automatically involving a termination proof. This involved construction assures logical safeness: in general, just adding axioms for recursive equations causes inconsistency for non-terminating functions. The resulting equations can now be used in the Isabelle simplifier.

The ML command. Isabelle itself is built on top of an SML execution environment, accessible with the ML command: ML{*`3 + 4`*} compiles "`3 + 4`", executes it, and displays the result in the output window. However, when so-called *code-antiquotations* such as `@{code NIL}` are used, the process is more involved because SML antiquotations implicitly refer to Isabelle values. Concretely, there is an additional processing step, resolving the needed Isabelle dependencies before the SML code is actually compiled. declare`[[ML_trace]]` shows this resolving step (varying depending on the antiquoted values). In Fig. 1, the two antiquotations `NIL` and `height` imply the following generated SML code:

```
structure Isabelle =
 struct
   datatype nat  = Zero_nat | Suc  of nat        ;
   datatype list = NIL      | CONS of nat * list ;
   fun height  NIL           = Zero_nat
     | height (CONS (x, t)) = Suc (height t)     ;
 end (*struct Generated_Code*)
```

This SML code reflects equivalently the one of Isabelle side (the datatype and fun declarations). During the compilation, antiquotations are then replaced by:

```
val NIL    = Isabelle.NIL
val height = Isabelle.height
```

which makes "`height NIL`" efficiently executable in the context of the compiled code — no symbolic representation is any longer involved.

Defining Isar Syntax. We are adding a new command Term in Fig. 1 with "`keywords` *Term*"; the *Isar*-component of Isabelle handling the "outer syntax" is in fact reconfigurable. Generally, any command from the Isabelle core APIs is accessible within the ML scope. So it is as well possible to implement Term for simulating datatype, fun or any existing Isabelle command.

In a nutshell, a combination of the techniques shown in this section will be used to construct our compiler in HOL, compile it to SML, and bind the compiled code to a USE tool-like syntax inside Isabelle/Isar.

2.2 UML/OCL: A Running Example

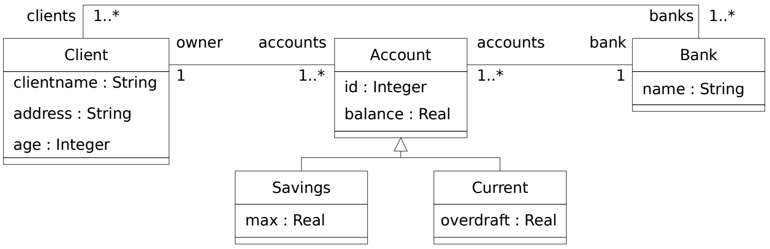


Fig. 2. A simple class model capturing a bank account

Let us consider here a small example of a UML class model together with its class invariants in OCL. The model of Figure 2 describes a set of clients owning bank accounts in different banks. Each account is either a current account or a savings account, and belongs to exactly one bank and one client. A client cannot have more than one current account in a given bank, but as many savings accounts as he likes. If a client is less than 25, his authorised overdraft is 250€ on every of his current accounts, otherwise no overdraft is allowed (it is set to 0). Moreover, the balance of a savings account must be between 0 and `max`. Finally, a consistency constraint has to be imposed: a client owning an account that belongs to a given bank must be a client of this bank.

3 Method: Tool-Construction by Reflection

Although it is perfectly feasible to program the compiler for class models to a corresponding datatype theory in SML , we choose to take even more advantage of the Isabelle framework instead. By using Isabelle/HOL as "implementation language" itself, we profit from the general proof-editing facilities as well as the possibility to *prove* properties over the compiler. For the moment, this covers only termination properties of the compiling functions, but the technique can in principle be used to prove complex meta-theoretic properties such as "if the class model is well-formed, the generated code will be type-correct wrt. HOL types".

The overall structure of the class-model compiler of Featherweight OCL is illustrated in Fig. 3. In the following, we will describe its components.

3.1 UML/OCL Meta-Model

Overall, the compiler has about 6000 lines of code - so we restrict ourselves to critical code-samples in our presentation. As short example, here is how we

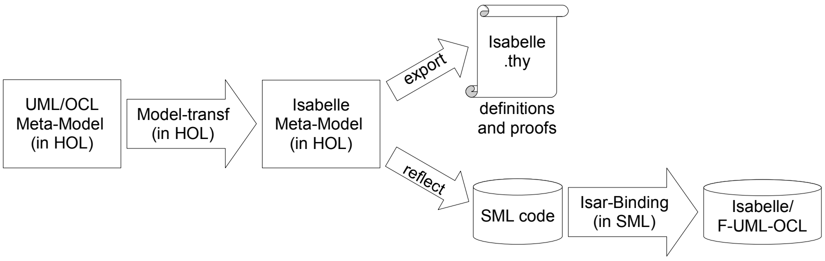


Fig. 3. Overview of the UML/OCL data-model compiler

define the Isabelle/HOL datatype behind the abstract syntax tree of the USE language, that we call the UML/OCL meta-model:

```
datatype uml_class = UmlClass
        string                                   (* name of the class    *)
      ( string (* name *) * uml_ty           ) list (* attribute         *)
      ( string (* name *) * uml_operation    ) list (* contract          *)
      ( string (* name *) * string           ) list (* invariant         *)
        string                                   (* link to superclasses *)
```

In this meta-model, a sample portion of our example Fig. 2 reads as follows:

```
[ UmlClass "Client" [ ( "clientname" , UmlType String),
                      ( "address"    , UmlType String) ] [] [] "OclAny",
  UmlClass "Bank"   [ ( "name"       , UmlType String) ] [] [] "OclAny",
  [...]                                                               ]
```

The compilation proceeds by elementary well-formedness checks and through a semantic elaboration. Finally, definitions for casts from `Client` to `OclAny` were produced, for example, defined in terms of the common data-universe for the given class model, together with proofs for cast-up-cast-down properties as mentioned before. For a complete list of definitions and lemmas, the reader is referred to [4].

3.2 Isabelle Meta-Model

While our meta model for Isabelle/HOL should be generic enough for representing all the concepts occurring in [4], we used a convenient abstraction wrt. to the *real* meta model defined in SML. For the moment, our simplistic abstraction is sufficiently expressive for modeling UML/OCL class models, although a long term goal would be the creation of symbolic tests generation procedures. Here is the Isabelle/HOL meta model describing Isabelle datatypes:

```
datatype hol_datatype =

   Datatype  string                                        (* name          *)
           ( string              (* name      *)
           * hol_simplety list   (* arguments *) ) list    (* constructors  *)
```

The previous meta model belongs to the following more general one, while a few is presented, there is actually an abundance of constructors:

```
datatype hol_theory = Theory_datatype  hol_datatype
                    | Theory_definition hol_definition
                    | Theory_lemma      hol_lemma
                    | [...]
```

3.3 Model-Transformation and Bindings to Isabelle/jEdit

The transformation from UML-Meta to Isabelle-Meta is purely defined in HOL (no tricks, no axioms, no SML), following linearly the structure of [4]. By applying this generic transformation to the bank example, we obtain automatically the definition of the object universe of [3] instantiated for the bank example: Fig. 4 (top) shows a simple fragment of datatypes (there are also proofs).

As described in Sec. 2.1, we now define a textual format for the UML/OCL meta-model and bind the generated ("reflected") compiler to it. Conceptually, it is similar to the Term command, only that UML data-models are simulated with the raw implementation of the datatype and lemma commands, instead of printing a term. We were inspired by the syntax of the USE tool[7] as entry point in Isabelle/jEdit. Finally, the interactive editing of the bank example leads to 2385 definitions and derived theorems, the source code is in Fig. 4 (bottom).

4 An Empirical Evaluation

This section gives some experimental results on run-time executions of the generated compiler. We study the following scenario: we build a sample of class models, where in each class model, every class inherits *explicitly* from one class, except `OclAny` standing as the only root: thus we have a tree where each class inherits *implicitly* from all classes present in the path of ancestors going to `OclAny`. For example, `Current` is an explicit subclass of `Account`, while `Current` also implicitly inherits from `OclAny`. Attributes and associations between classes are not considered here; as a shorthand, the "subclass" word alone will designate an explicit subclass.

We present Fig. 5 a table reporting the number of theorems associated to each tested class model. Numbers of generated theorems are indicated by powers of 1000 (so Kilo and Mega), and those in italic are an estimation based on the size of the generated file. The class models we are measuring can be identified uniquely by pairs (X, Y) where X is the exact number of subclasses of every class having at least one subclass; and Y is the depth of the inheritance tree (without `OclAny`, so the minimum is one for a tree containing at least `OclAny` with another element). Class-models appear sorted in the table according to the following priority: 1) by row using the total number of classes in the class model, c represents the cardinal without `OclAny`; then 2) by column using the depth of the inheritance tree. For instance, class models in the row $[(1, 30), (2, 4), (5, 2), (30, 1)]$ are sorted in decreasing order by depth, all having 31 classes ($c = 30$, `OclAny` counts for 1).

Since the generated UML/OCL theorems serve for solving (the most automatically) UML/OCL formulas manually entered by the user, our goal is to continue

$$
\begin{array}{lll}
\text{datatype } \mathrm{type}_{\mathrm{Savings}} & = & \mathrm{mk}_{\mathrm{Savings}}\ \mathrm{oid}\ \mathrm{int}_{\bot}\ \mathrm{real}_{\bot} \\
\text{datatype } \mathrm{typeoid}_{\mathrm{Savings}} & = & \mathrm{mkoid}_{\mathrm{Savings}}\ \mathrm{type}_{\mathrm{Savings}}\ \mathrm{real}_{\bot} \\
\text{datatype } \mathrm{type}_{\mathrm{Account}} & = & \mathrm{mk}_{\mathrm{Account\_Savings}}\ \mathrm{typeoid}_{\mathrm{Savings}} \\
 & \mid & \mathrm{mk}_{\mathrm{Account\_Checks}}\ \mathrm{typeoid}_{\mathrm{Checks}} \\
 & \mid & \mathrm{mk}_{\mathrm{Account}}\ \mathrm{oid} \\
\text{datatype } \mathrm{typeoid}_{\mathrm{Account}} & = & \mathrm{mkoid}_{\mathrm{Account}}\ \mathrm{type}_{\mathrm{Account}}\ \mathrm{int}_{\bot}\ \mathrm{real}_{\bot} \\
\text{datatype } \mathrm{type}_{\mathrm{OclAny}} & = & \mathrm{mk}_{\mathrm{OclAny\_Client}}\ \mathrm{typeoid}_{\mathrm{Client}} \\
 & \mid & \mathrm{mk}_{\mathrm{OclAny\_Bank}}\ \mathrm{typeoid}_{\mathrm{Bank}} \\
 & \mid & \mathrm{mk}_{\mathrm{OclAny\_Account}}\ \mathrm{typeoid}_{\mathrm{Account}} \\
 & \mid & \mathrm{mk}_{\mathrm{OclAny\_Savings}}\ \mathrm{typeoid}_{\mathrm{Savings}} \\
 & \mid & \mathrm{mk}_{\mathrm{OclAny\_Checks}}\ \mathrm{typeoid}_{\mathrm{Checks}} \\
 & \mid & \mathrm{mk}_{\mathrm{OclAny}}\ \mathrm{oid} \\
\text{datatype } \mathrm{typeoid}_{\mathrm{OclAny}} & = & \mathrm{mkoid}_{\mathrm{OclAny}}\ \mathrm{type}_{\mathrm{OclAny}}
\end{array}
$$

```
Class Bank
      Attributes
        name        : String
End

Class Client
      Attributes
        clientname  : String
        address     : String
        age         : Integer
End

Class Account
      Attributes
        id          : Integer
        balance     : Real
End

Class Savings < Account
      Attributes
        max         : Real
End
```

```
Class Current < Account
      Attributes
        overdraft   : Real
End

Association clients
  Between Bank    [1 .. *]
            Role banks
          Client  [1 .. *]
            Role clients    End

Association accounts
  Between Account [1 .. *]
            Role accounts
          Client  [1]
            Role owner      End

Association bankaccounts
  Between Account [1 .. *]
            Role accounts
          Bank    [1]
            Role bank       End
```

```
Context c: Savings
  Inv A : ‘0 < (c .max)‘
  Inv B : ‘c .balance <= (c .max) and 0 <= (c .balance)‘

Context c: Current
  Inv A : ‘25 < (c .owner .age) implies (c .overdraft = 0)‘
  Inv B : ‘c .owner .age <= 25 implies (c .overdraft = -250)‘

Context c: Client
  Inv A : ‘c .accounts ->collect(banks) = c .banks‘
```

Fig. 4. Modeling the bank account in UML/OCL with Isabelle/jEdit

c	depth c	depth 5	depth 4	depth 3	depth 2	depth 1
12	$(1, c)$ 14K				$(3, 2)$ 12K	$(c, 1)$ 11K
14	$(1, c)$ 20K			$(2, 3)$ 17K		$(c, 1)$ 16K
20	$(1, c)$ *52K*				$(4, 2)$ *39K*	$(c, 1)$ *39K*
30	$(1, c)$ *155K*		$(2, 4)$ *121K*		$(5, 2)$ *115K*	$(c, 1)$ *115K*
39	$(1, c)$ *330K*			$(3, 3)$ *240K*		$(c, 1)$ *240K*
42	$(1, c)$ *409K*				$(6, 2)$ *288K*	$(c, 1)$ *294K*
56	$(1, c)$ *964K*				$(7, 2)$ *649K*	$(c, 1)$ *661K*
62	$(1, c)$ *1.3M*	$(2, 5)$ *907K*				$(c, 1)$ *882K*
72	$(1, c)$ *2M*				$(8, 2)$ *1.3M*	$(c, 1)$ *1.3M*
84	$(1, c)$ *3.3M*			$(4, 3)$ *2.1M*		$(c, 1)$ *2.1M*
90	$(1, c)$ *4.2M*				$(9, 2)$ *2.5M*	$(c, 1)$ *2.5M*

Fig. 5. Number of theorems generated

to generate new UML/OCL theorems, while keeping an objective of shortening the size of proofs whenever applicable. In a class model with only `OclAny` as class, we obtain a theory comprising 182 theorems proven automatically; by adding another class, we reach 384 theorems.

Besides the constraint on the generation of a high number of theorems, time or space involved for performing the generation is obviously a criteria to consider as enhancement: we need at least 9G of RAM memory for generating in 1 min one of the three examples of $c = 56$ classes. For $c = 90$, 28G becomes mandatory to be executed in 7 min; however, there are substantial potentials for optimization.

5 Conclusion

We have shown a method to construct semantic-based tools for textual domain-specific languages (DSLs) based on UML and OCL. Based on Isabelle/HOL theories that capture the semantic essence of a DSL (in our case: class-models plus OCL invariants and contracts), we describe model-transformation from a formal UML meta-model to an Isabelle meta-model that generates the necessary properties automatically. Compared to conventional implementations of *code-generators* for OCL, the resulting tool is clearly not competitive in terms of compilation size of models, essentially because each theorem is complemented with a proof of a certain size. On the other hand our tool is unique that it actually generates a large number of theorems resulting from class-models which are necessary for symbolic execution and UML/OCL interactive proving. Moreover, our tool — for which we still see a large potential of optimization — can serve as reference environment for the UML/OCL language.

Related Work. The idea to use SML for supporting data-type theories is in itself very old and deeply linked from the very beginning with theorem proving environments such as Edinburgh-LCF, HOL4, HOL-light, Isabelle and Coq. The application to *object-oriented* data-type theories is also not new — earlier works in this line are [11], for example. In contrast to [5], we applied these techniques to UML under *closed-world assumption* for a standard-conform 4-valued logics for OCL, which is seen as the semantic framework for DSL's. This is particularly important and challenging since heterogenous system specifications need to be combined in a seamless way, and since semantically correct tools have to be developed for these language combinations.

Future Work. It is our ultimate goal to develop the technology up to the point that it can be used for automated test-generation following the lines of [2]. We expect that the approach can be applied to textually presented sequence models, state-machines or MARTE-profiles.

Acknowledgments. This research work has been carried out under the leadership of the Technological Research Institute SystemX, and therefore granted with funds of the Program "Investissements d'Avenir" (`www.irt-systemx.fr`).

References

[1] U. Aßmann, A. Bartho, C. Bürger, S. Cech, B. Demuth, F. Heidenreich, J. Johannes, S. Karol, J. Polowinski, J. Reimann, J. Schroeter, M. Seifert, M. Thiele, C. Wende, and C. Wilke. Dropsbox: the dresden open software toolbox. *Software & Systems Modeling*, 13(1):133–169, 2014.

[2] A. D. Brucker, M. P. Krieger, D. Longuet, and B. Wolff. A specification-based test case generation method for UML/OCL. In J. Dingel and A. Solberg, editors, *MoDELS Workshops*, number 6627, pages 334–348. 2010. Selected best papers from all satellite events of the MoDELS 2010 conference. Workshop on OCL and Textual Modelling.

[3] A. D. Brucker, D. Longuet, F. Tuong, and B. Wolff. On the semantics of object-oriented data structures and path expressions. In J. Cabot, M. Gogolla, I. Rath, and E. Willink, editors, *Workshop on OCL, Model Constraint and Query Languages (OCL 2013)*, 2013. An extended version of this paper is available as **lri!** Technical Report 1565.

[4] A. D. Brucker, F. Tuong, and B. Wolff. Featherweight OCL: A proposal for a machine-checked formal semantics for OCL 2.5. *Archive of Formal Proofs*, Jan. 2014. `http://afp.sf.net/entries/Featherweight_OCL.shtml`, Formal proof development.

[5] A. D. Brucker and B. Wolff. An Extensible Encoding of Object-oriented Data Models in HOL with an Application to IMP++. *Journal of Automated Reasoning (JAR)*, Selected Papers of the AVOCS-VERIFY Workshop 2006(3–4):219–249, 2008. Serge Autexier, Heiko Mantel, Stephan Merz, and Tobias Nipkow (eds).

[6] A. D. Brucker and B. Wolff. Featherweight OCL: A study for the consistent semantics of OCL 2.3 in HOL. In *Workshop on OCL and Textual Modelling (OCL 2012)*, pages 19–24, 2012. The semantics for the Boolean operators proposed in this paper was adopted by the OCL 2.4 standard.

[7] M. Gogolla, F. Büttner, and M. Richters. USE: A UML-based specification environment for validating UML and OCL. *Sci. Comput. Program.*, 69(1-3):27–34, 2007.

[8] T. Parr. *The Definitive ANTLR Reference Guide: Building Domain-Specific Languages.* Pragmatic Programmers, 2007.

[9] D. Steinberg, F. Budinsky, M. Paternostro, and E. Merks. *EMF: Eclipse Modeling Framework, 2nd Edition.* Eclipse Series. Addison-Wesley Professional, Dec 16, 2008.

[10] The Object Management Group (OMG). The MOF and OMG's model driven architecture (MDA). Available at `http://www.omg.org/mof` (2014).

[11] M. Wenzel. Type classes and overloading in higher-order logic. In *TPHOLs*, pages 307–322, 1997.

[12] E. Willinks. The Eclipse OCL project. Available at `http://www.willink.me.uk` (2014).

Realizing Model Simplifications with QVT Operational Mappings

Alexander Kraas

Poppenreuther Str. 45, D-90419 Nürnberg, Germany
alexander.kraas@gmx.de

Abstract. After parsing the input of a textual modeling language, further processing steps may be required before the result can be mapped to corresponding elements in a model. For instance, such a processing step can be the simplification of syntactic constructs. An approach for model simplification resting on transformation patterns is presented in this paper. The presented approach rests on the refinement of a derived base transformation with the superimposition of mapping operations. The transformations are specified with the Operational Mappings part of the Query/View/Transformations (QVT) specification.

Keywords: QVT-O, Transformation, Patterns, Simplification

1 Introduction

In general, modeling languages can be classified into textual as well as graphical modeling languages. However, also hybrid approaches using both kinds of modeling exist. For instance, the Specification and Description Language (SDL) [9] can be considered as such a language. Further processing steps (e.g. simplifications) may be required after the textual input of a hybrid modeling language is parsed and represented in terms of a Concrete Syntax Tree (CST). According to the taxonomy given in [1], model simplification is an approach where particular syntactic constructs are transformed into more primitive constructs. Usually, dedicated frameworks for language transformations, such as Stratego/XT [7], can be utilized for this purpose. However, an access to an already existing model may be required for the simplification of the textual input of a hybrid language. If a CST is defined in terms of a metamodel, the Query/View/Transformations (QVT) [8] specification supports such an transformation scenario.

Since the general realization of transformation patterns by using the Relational Language of QVT is already discussed in [3], in this paper the implementation of patterns for model simplification by using the Operational Mappings of QVT is discussed. The QVT features transformation extension and superimposition of mapping operations play a key role for the presented approach. Even if these features are defined in the QVT specification [8], only less information concerning their combined application can be found in the literature. Hence, the successful application of the approach for the textual SDL editor of the SU-MoVal framework [10] is used in this paper to illustrate the implementation of simplification patterns. The transformations within the SU-MoVal framework are executed by the QVT operational component (QVTo) [12] of Eclipse.

The rest of this paper is structured as follows. The proposed approach and transformation patterns for model simplification are introduced in section 2. Related work is discussed in section 3 and a conclusion is given in section 4.

2 Transformation Patterns for Model Simplification

An approach to implement patterns for model simplification with QVT Operational Mappings (QVT-O) is presented in this chapter. After the general approach is discussed, a transformation example is introduced in section 2.2, which is used to explain the approach more descriptive in subsequent sections. Finally, the realization of exemplary patterns for model simplification is discussed in section 2.4.

2.1 General Approach

The presented approach to simplify models by using QVT Operational Mappings (QVT-O) rests upon a common endogenous base transformation (T_{CB}) that is extended by another transformation (T_{SP}) implementing a particular simplification pattern.

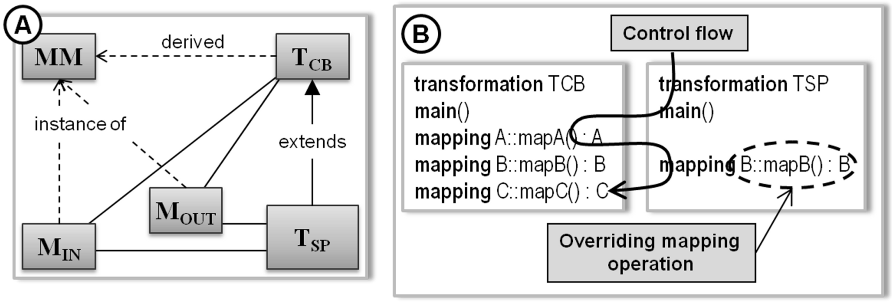


Fig. 1. Conceptual overview

Derivation of a common base transformation: As shown in part A of Fig. 2, the required transformation T_{CB} is derived from a metamodel (MM) and the input model (M_{IN}) and the output model (M_{OUT}) are instances of MM. The purpose of T_{CB} is to produce only a one-to-one copy of model M_{IN} and therefore, a particular mapping operation for each metaclass is contained in T_{CB}. Even if a higher-order transformation could be applied to derive T_{CB} from a metamodel MM, transformation T_{CB} is generated with a Model-to-Text (M2T) approach so that its source code can be partially reused to specify transformation T_{SP}.

In order to achieve that transformation T_{CB} makes a copy of M_{IN}, all elements of M_{IN} are mapped to equal elements in M_{OUT}. Hence, T_{CB} implements an entire rewrite system that consists of contextual mapping operations that invoke other mapping operations in a nested manner to map the properties of an input element.

Realizing a simplification pattern: When a model simplification pattern is implemented by transformation T_{SP}, the control flow of T_{CB} has to be redirected at points where the relevant elements to be simplified occur. Usually, such points are mapping operations that are defined in the context of the relevant metaclasses. After the mapping operations in T_{CB} are identified, they have to be overridden by corresponding mapping operations (see part B of Fig. 2) in T_{SP}, which implement the desired simplification pattern.

2.2 Transformation Example and Required Metaclasses.

In order to make the approach discussed in the following sections more descriptive, a transformation example resting on a metamodel for representing parse-trees of the concrete syntax (`CS`) of the Specification and Description Language (SDL) [9] is used. This example is taken from the SDL-UML Modeling and Validation framework (SU-MoVal) [10].

Since the concrete syntax of SDL makes use of so-called short hand notations that have to be transformed to equivalent language constructs, this is considered to be a good case study for model simplification. In particular, the transformation example consists of the transformation of a mathematical or logical in-line operator to a corresponding operation application. For instance, the concrete syntax expression `1 + 1` is transformed to `"+"(1, 1)`.

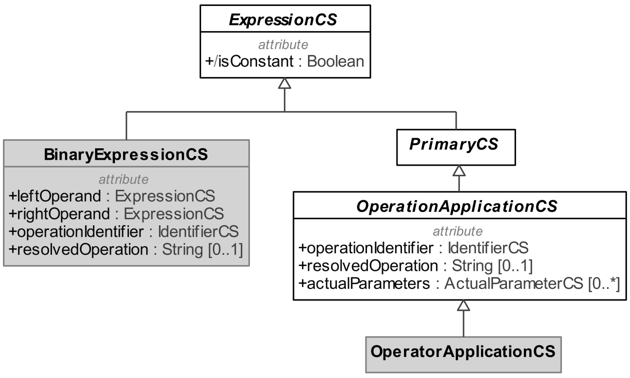

ExpressionCS
attribute
+/isConstant : Boolean
BinaryExpressionCS
attribute
+leftOperand : ExpressionCS
+rightOperand : ExpressionCS
+operationIdentifier : IdentifierCS
+resolvedOperation : String [0..1]
PrimaryCS
OperationApplicationCS
attribute
+operationIdentifier : IdentifierCS
+resolvedOperation : String [0..1]
+actualParameters : ActualParameterCS [0..*]
OperatorApplicationCS

Fig. 2. Binary expression metaclass

As shown in Fig. 2, the `BinaryExpressionsCS` metaclass is a sub-type of `ExpressionCS` and it is used to represent mathematical or logical inline operators (e.g. + operator) within SDL expressions. After application of the transformations patterns discussed in the subsequent sections, an instance of `BinaryExpressionCS` is transformed to an `OperatorApplicationCS`.

2.3 Derivation of a Common Base Transformation

The derivation of a transformation from a metamodel by using a Model-to-Text (M2T) transformation is out of scope of this paper. Hence, the derivation approach is only discussed on a high-level of abstraction. For instance, the M2T tool Acceleo [11] is used for the derivation task during the implementation of SU-MoVal [10].

Transformation Main Part. As shown in the code example below, the required `main` operation of T_{CB} is empty, because the concrete behavior is specified by a transformation that extends T_{CB}. Furthermore, the input and output parameters of T_{CB} have the same type because T_{CB} is an endogenous transformation.

```
modeltype CS uses ConcreteSyntax ("http://...");
transformation TCB(in inp:CS, out outp:CS)
main() { // Nothing to do here !}
```

Mapping Operations for Metaclasses. For each metaclass of the example metamodel `CS`, a corresponding mapping operation for T_{CB} is derived. Depending on the kind of a metaclass (abstract or non-abstract), two different kind of mapping operations are generated. In addition, the context (`self`) and the `result` type of a mapping operation correspond to the currently processed metaclass.

For each abstract metaclasses, a disjunctive mapping operation is introduced consisting of an ordered list of mapping operations for all subclasses of that metaclass (e.g. `ExpressionCS`). An important fact is that attributes, if any, of an abstract metaclass cannot taken into account by a disjunctive mapping operation. Instead, these attributes are processed by each mapping operation listed as disjunctive alternative.

```
mapping CS::ExpressionCS::mapExpressionCS() : CS::ExpressionCS
disjuncts
  CS::EqualityExpressionCS::mapEqualityExpressionCS,
  CS::PrimaryCS::mapPrimaryCS,
  CS::TypeCoercionCS::mapTypeCoercionCS,
  CS::MonadicExpressionCS::mapMonadicExpressionCS,
  CS::CreateExpressionCS::mapCreateExpressionCS,
  CS::RangeCheckExpressionCS::mapRangeCheckExpressionCS,
  CS::BinaryExpressionCS::mapBinaryExpressionCS {}
```

In contrast to abstract metaclasses, all owned and inherited attributes of a non-abstract metaclass (e.g. `BinaryExpressionCS`) are processed for the derivation of the body of a corresponding mapping operation.

```
mappingCS::BinaryExpressionCS::mapBinaryExpressionCS()
 : CS::BinaryExpressionCS {
   result.resolvedOperation := self.resolvedOperation;
   result.operationIdentifier := self.operationIdentifier.map
      mapIdentifierCS();
   result.leftOperand  := self.leftOperand.map mapExpressionCS();
   result.rightOperand := self.rightOperand.map mapExpressionCS();
   result.resolvedType := self.resolvedType;
}
```

Processing of Metaclass Attributes. In general, the mapping operation for a non-abstract metaclass assigns each owned or inherited attribute of an input object (`self`) to a corresponding attribute of the result object (`result`). Since QVT-O distinguish between assignments to mono- or multi-valued properties or variables, this has also to be considered during the processing of attributes. Hence, a simple assignment (`':='` operator) is used for the mapping of an attribute with an upper multiplicity of 1. In contrast, attributes with an upper multiplicity of >1 are assigned with a composite assignment (`'+='` operator).

```
result.myProperty := self.myProperty.map mapMySubclassA();
result.myProperty += self.myProperty->map mapMySubclassA();
```

The attribute type is another property that is taken into account during the derivation process. Usually, a type dependent mapping operation is invoked, before the value is assigned to the output object. This is not the case for attributes with a primitive (e.g. `String`) or an enumerated type for which the input value is just copied to the corresponding attribute of the output object.

```
result.myProperty := self.myProperty;
result.myProperty += self.myProperty;
```

Design Principles and Used Features of QVT-O. Apart from the discussed extension mechanism also the access mechanism of QVT-O could be considered to access mapping operations of T_{CB}. However, this mechanism does not support transformation inheritance so that the transformation control flow cannot be redirected through superimposition of mapping operations of T_{SP}.

Furthermore, QVT-O supports the inheritance and the merge of mapping operations. These features where not taken into account for the presented approach, because they induce a combined execution of all involved mapping operations. If these features would be used, the mapping rules for a particular metaclass had to be specified in different mapping operations. Instead, the discussed approach recommends to process all inherited and owned attributes of a metaclass within one mapping operation of T_{CB}. The advantage for the implementation of transformation patterns is a better maintainability and a lower complexity, because required mapping rules can be specified within one overriding mapping operation of T_{SP}.

Since the inheritance and the merge of mapping operations are not used, also the QVT-O feature of abstract mapping operations is not required in order to process abstract metaclasses of MM. Instead, disjunctive mapping operations are used for this purpose, because they make a fine grained manipulation of the transformation's control flow possible.

2.4 Exemplary Transformation Patterns for Model Simplification

Resting on the already introduced approach (see Sec. 2.1), exemplary transformation patterns for model simplification are discussed in more detail in this section. For this purpose, the transformation example of section 2.2 is used in the following paragraphs.

Top-level Simplification. A model can be represented as a tree structure of nested model elements, which are instances of different metaclasses of a metamodel. The most simple use case for model simplification is the transformation of top-level elements only. If this pattern is applied to the transformation example (see Sec. 2.2), only the top-level instances of `BinaryExpressionCS` will be transformed to a corresponding `OperatorApplicationCS` and nested `BinaryExpressionsCS` will be preserved (see example output O_1 shown in Fig. 3).

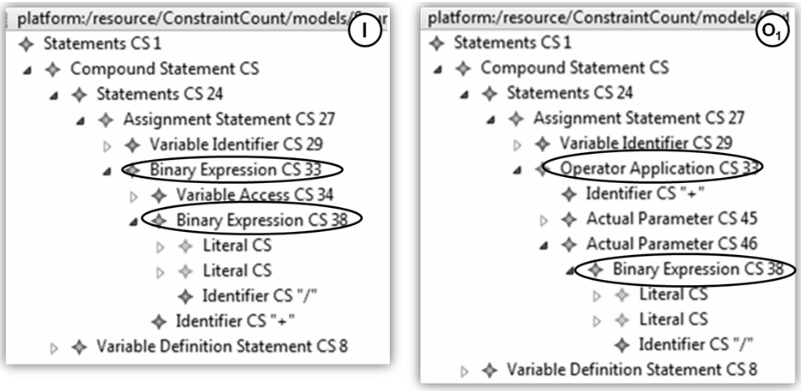


Fig. 3. Example input model and output model 1

In order to realize this pattern for the given transformation example with QVT, the transformation T_{CB} is extended by transformation `InfixOperationToOperator`. Within the `main()` method of this transformation, the root element of the model is selected and an appropriate mapping operation of the $\mathrm{T_{CB}}$ transformation is invoked. According to the presented approach, the control flow of the transformation remains in $\mathrm{T_{CB}}$, as long as one of its mapping operations is overridden by another operation specified in the `InfixOperationToOperator` transformation.

```
transformation InfixOperationToOperator
  (in input : CS, out output : CS) extends TCB;
main() {
  input.rootObjects()[CS::StatementCS]->map mapStatmentCS(); }
```

In order to transform each occurrence of `BinaryExpressionsCS`, the control flow within $\mathrm{T_{CB}}$ is redirected at points where associated mapping operations are invoked. Therefore, the operation `mapExpressionCS()` is overridden as follows:

```
mapping CS::ExpressionCS::mapExpressionCS() : CS::ExpressionCS
disjuncts
  CS::EqualityExpressionCS::mapEqualityExpressionCS,
  CS::PrimaryCS::mapPrimaryCS,
  CS::TypeCoercionCS::mapTypeCoercionCS,
  CS::MonadicExpressionCS::toOperatorApplication,
  CS::CreateExpressionCS::mapCreateExpressionCS,
  CS::RangeCheckExpressionCS::mapRangeCheckExpressionCS,
  CS::BinaryExpressionCS::toOperatorApplication {}
```

The last line (bold printed) of `mapExpressionCS()` is modified in comparison to the overridden operation in $\mathrm{T_{CB}}$. That is because the control flow shall be redirected to the `toOperatorApplication()` operation that implements the mapping of a `BinaryExpressionCS` to an `OperatorApplicationCS`.

```
mappingCS::BinaryExpressionCS::toOperatorApplication()
  : CS::OperatorApplicationCS {
  result.resolvedType := self.resolvedType;
  result.resolvedOperation := self.resolvedOperation;
  result.operationIdentifier := self.operationIdentifier;
  result.actualParameters := OrderedSet {
    object ActualParameterCS { expression := self.leftOperand },
    object ActualParameterCS { expression := self.rightOperand }};
}
```

Since only top-level instances of `BinaryExpressionCS` shall be transformed by the `toOperatorApplication()` operation, all attributes of an input element are assigned directly to the created output element (without invoking any other mapping operation).

Recursive Simplification. The objective of the pattern for recursive simplification is to transform all model elements of a particular kind. This is realized with a recursive call of the mapping operation, if required. After applying a recursive simplification pattern for the given transformation example, the input model `I` (shown in Fig. 3) is transformed to the output model `O2` (see Fig. 4). As expected, all instances of `BinaryExpressionCS` in the output model `O2` are transformed to instances of `OperatorApplicationCS`.

In order to implement the recursive transformation pattern for the given transformation example, a slightly modified variant of the already discussed `InfixOperationToOperator` transformation is used. Hence, the `toOperatorApplication()` mapping operation is modified in a way so that also for nested elements dedicated mapping operations are invoked instead of copying them.

```
result.actualParameters :=  OrderedSet {
  object ActualParameterCS { expression :=
    self.leftOperand.map mapExpressionCS() },
  object ActualParameterCS { expression :=
    self.rightOperand.map mapExpressionCS() } }
 };
```

The additional map operation calls (bold printed) in the code snippet shown above realize the recursive call to the `toOperatorApplication()` mapping operation. When one of the `mapExpressionCS()` operations is invoked, appropriate mapping operations defined in T_{CB} are processed as long as an instance of `BinaryExpressionCS` shall be mapped. If this is the case, the control flow of the transformation is redirected to the `toOperatorApplication()` operation once again.

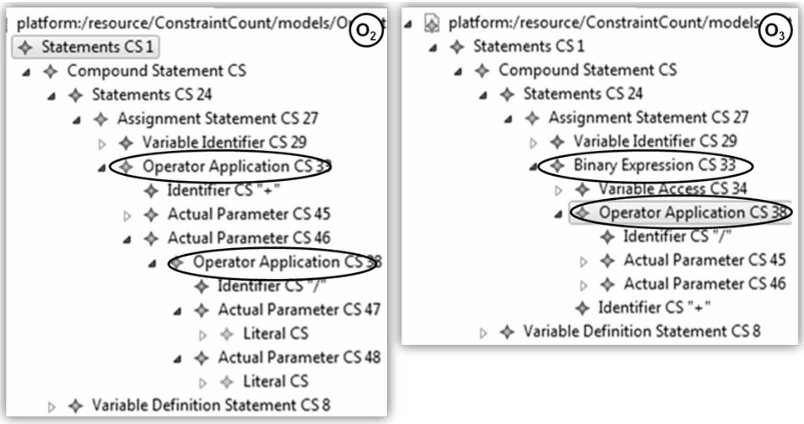


Fig. 4. Output models 2 and 3

Simplification of Leaf Elements. The transformation pattern for simplification of leaf elements can be considered as opposite of the already presented top-level simplification pattern. A simplification of leaf elements applied to the transformation example causes a mapping of leaf `BinaryExpressionCS` elements to corresponding `OperatorApplicationCS` elements, whereas top-level elements of that kind are only copied to the output model (see output model `O3` in Fig. 4).

A modified variant of the `InfixOperationToOperator` transformation can be used to implement the simplification of leaf elements. Therefore, the `mapBinaryExpressionCS()` operation, which just makes a copy of the input element, is added as an additional mapping alternative to the disjunctive mapping operation `mapExpressionCS()`. It is important to add `mapBinaryExpressionCS()` after `toOperationApplication()`, because the disjunctive alternatives are processed in sequential order.

```
mapping CS::ExpressionCS::mapExpressionCS() : CS::ExpressionCS
disjuncts
  ...
  CS::BinaryExpressionCS::toOperatorApplication,
  CS::BinaryExpressionCS::mapBinaryExpressionCS {}
```

A **when** clause is added to the `toOperatorApplication()` operation, because it shall only be invoked for input elements that have no further nested `BinaryExpressionCS` elements. If the condition of the when clause is not fulfilled, the mapping operation is not invoked.

```
mappingCS::BinaryExpressionCS::toOperatorApplication()
  : CS::OperatorApplicationCS
when { self.allSubobjectsOfType
        (CS::BinaryExpressionCS)->size() = 0 }
```

3 Related Work

Many works [2, 3, 6] concerning the refinement of endogenous transformations and the implementation of transformation patterns with the QVT Relational Language (QVT-R) exist. In addition, the usage of the AtlanMod Transformation Language (ATL) for this purpose is analysed in [4, 6]. Even if the mentioned works discuss possible approaches, the analysis of the ATL Transformation Zoo in [5] comes to the conclusion that transformation superimposition is rarely used in practice.

In contrast to before mentioned works, the approach presented in this paper rests on the QVT Operational Mappings (QVT-O) [8], which is an imperative transformation language and not a relational language as used by the mentioned works. Furthermore, instead of introducing only a copy pattern for QVT-O transformations, the presented work discuss the refinement of generated copy patterns towards essential patterns for model simplification.

4 Conclusion

The approach for model simplification by using the QVT Operational Mappings presented in this paper is not only restricted to the used metamodel of the textual notation of SU-MoVal [10]. That is because the rules for the derivation of a common base transformation, which is the starting point for all discussed simplification patterns, are also applicable to each other kind of metamodel.

In addition, the applicability of the approach is not limited to the patterns discussed in the section before, because further patterns for simplification can be realized in a similar manner. As a matter of principle, also other kind of endogenous model transformations could be implemented in the same manner, because the area of model simplification is only used as an exemplarily use-case in order to demonstrate the overall approach. A running transformation example and further information can be obtained from the SU-MoVal homepage [10].

References

1. Mens T., Van Gorp P.: A Taxonomy of Model Transformation. In: Electronic Notes in Theoretical Computer Science, vol. 152, pp. 125 – 142. Elsevier, Amsterdam (2006)
2. Goldschmidt T., Wachsmuth G.: Refinement transformation support for QVT Relational transformations. In: 3rd Workshop on Model Driven Software Engineering, MDSE 2008 (2008)
3. Iacob M-E., Steen M., Heerink L.: Reusable model transformation patterns. Enterprise Distributed Object Computing Conference Workshops, 2008 12th., pp. 1 – 12. IEEE Press, New York (2008)
4. Tisi M., et. al.: Refining Models with Rule-based Model Transformations. INRIA, Rennes, France (2011)
5. Kusel A., et. al.: Reality Check for Model Transformation Reuse: The ATL Transformation Zoo Case Study. In: 2nd Workshop on the Analysis of Model Transformations, AMT@Models'13, pp. 1 – 10. Miami (2013)
6. Wagelaar D., Van Der Straeten R., Deridder D.: Module superimposition: a composition technique for rule-based model transformation languages. In: Software & Systems Modeling, vol. 9, issue 3, pp. 285 – 309, Springer, Heidelberg (2010)
7. Bravenboer M., Kalleberg K. T., Vermaas R., Visser E.: Stratego/XT 0.17. A language and toolset for program transformation. In: Science of Computer Programming, vol. 72, issue 1, pp. 52 – 70, Elsevier (2008)
8. Object Management Group: Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification. Version 1.1. OMG Document Number: formal/2011-01-01. http://www.omg.org/spec/QVT/1.1/PDF/
9. International Telecommunication Union: Recommendation Z.101 (12/2011), Specification and Description Language – Basic SDL-2010. https://www.itu.int/rec/TREC-Z.101/en
10. SDL-UML Modeling and Validation (SU-MoVal) framework, Open source software, http://www.su-moval.org/
11. Acceleo Model-to-Text transformation framework, Open source software, http://www.eclipse.org/acceleo/
12. QVT Operational component of Eclipse, Open source software, http://projects.eclipse.org/projects/modeling.mmt.qvt-oml

CdmCL, a Specific Textual Constraint Language for Common Data Model

Ahmed Ahmed, Paola Vallejo, Mickaël Kerboeuf, Jean-Philippe Babau

Lab-STICC / UBO / UEB, Brest, France
alahmed4ever@yahoo.com,{vallejoco,kerboeuf, babau}@univ-brest.fr

Abstract. Common Data Model is an abstract data model for scientific datasets that can be constrained by OCL. To hide complexity of OCL, *CdmCL* is proposed as a specific textual constraint language for CDM. *CdmCL* is based on the CDM structure and results in a set of constraint categories. *CdmCL* provides a user-friendly front end in order to define constraints which are subsequently translated to OCL. The conformity tool is based on an existing OCL checker integrated in EMF. *CdmCL* is experimented on the OceanSITES standard.

Keywords: OCL, common data model, conformity, constraint generation

1 Introduction

To improve interoperability, scientific dataset modeling follows standards like Unidata's Common Data Model (CDM) [1]. Since CDM is a general purpose model, scientists use specific standards like OceanSITES [2] for specific data modeling. A standard defines a set of additional constraints, classically expressed in a natural language. To check if CDM data conforms to a standard, like OceanSITES, a code-oriented checker is used classically. Thus, the constraints are not formalized and a modification in the standard results in a manual modification of the code.

To handle the problems of a code-centric approach, constraints can be implemented using Object Constraints Language (OCL) [3]. OCL is a formal language that significantly improves the clarity of models and makes them more precise [4]. But unfortunately, it is difficult to write correct OCL statement as many OCL constraints results in inaccurate and erroneous constraints [5], [6].

In this paper, we propose a textual domain specific constraint language *CdmCL* to reduce the complexity of handling OCL syntax. *CdmCL* is dedicated for scientific data standards. It is based on a set of constraints categories deduced from the CDM structure. Then, OCL constraints are generated and used by an OCL checker integrated in the Eclipse Modeling Framework (EMF) [7].

The paper is organized as follows. In the first section, CDM is introduced. Then OceanSITES is presented as a motivating example. *CdmCL* and the conformity tool generation are then presented before to be evaluated on OceanSITES. Before to conclude, related works are discussed.

2 Common Data Model

Unidata’s Common Data Model (CDM) is an abstract data model for scientific datasets. It is based on three layers, data access layer, coordinate system layer and scientific feature type layer. Our work focuses on data access layer also called syntactic layer that handles data modeling part. The complete data model and detailed description is given in [1]. The main classes (see Figure 1) are:

- *DataSet*: a file, such as NetCDF file, characterized by a file name (*location*).
- *Group*: a container for *dimensions*, *attributes*, *variables* and nested *subGroups*.
- *Dimension*: the array shape of a *Variable*, characterized by a *name* and a *length*;
- *Variable*: a container for data, characterized by a *name*, a *dataType*, a set of *dimensions* that define its array shape, and optionally a set of *attributes*.
- *Attribute*: a metadata to characterize a *Variable* or a *Group*, characterized by a *name*, a *dataType* and a *value*.

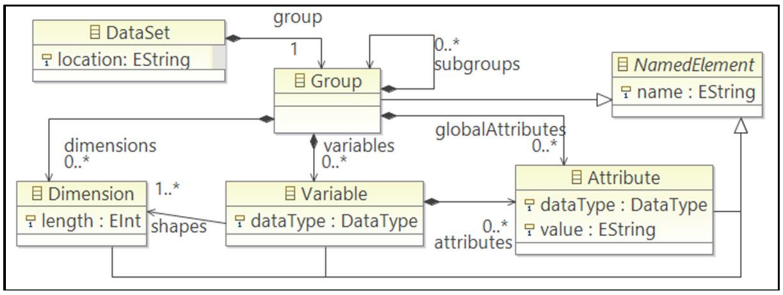


Fig. 1. An excerpt of the Common Data Model ecore (CDM.ecore)

3 OceanSITES

OceanSITES [2] is a worldwide system for gathering and measuring scientific data especially for time series sites, called ocean reference stations. It conforms to the CDM model but with some constraints. The OceanSITES User Manual holds around 30 pages of constraints expressed in natural language. These constraints are of different forms: naming conventions, possible attribute values, constraints on dimension length and many others. As example, a DataSet should hold instances of Dimension called *TIME*, *LATITUDE*, *LONGITUDE*, instances of Attributes called *data_type* and *format_version*, and an instance of Variable called *TIME*. The variable *TIME* should be of *double* datatype. Figure 2 illustrates a small excerpt of OceanSITES standard from the manual to the left and a small excerpt of CDM data respecting the OceanSITES standard to the right.

To check the data conformity to OceanSITES, a Java tool already exists [8]. Since constraints are not formalized and the tool is hand-coded, there is no guaranty on checking. Furthermore, for each different data format, a particular tool should be developed. To avoid constraint edition ambiguity and to reduce conformity tool development, we present now the *CdmCL* language.

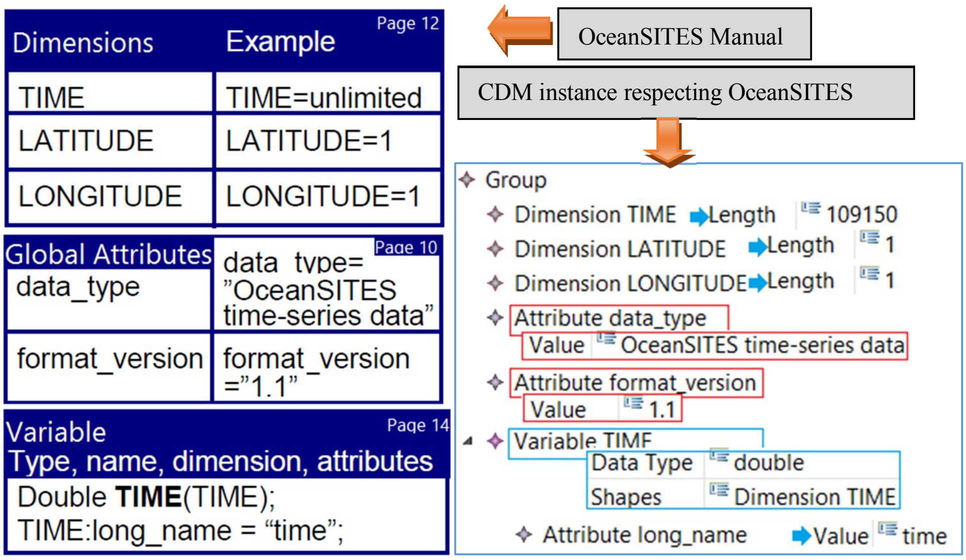


Fig. 2. Excerpts of OceanSITES manual and CDM instance

4 *CdmCL* Language

4.1 Concept References

This work focuses on the automatic generation of OCL constraints from *CdmCL*. On the one hand, the *CdmCL* front end needs to be human readable and close to the classical standard. Therefore, Xtext has been used to define the textual grammar. On the other hand, each CdmCL concept has a semantic expressed using OCL.

In standards, most of the constraints are related to a specific instance of a named CDM concept (Variable, Dimension and Attribute): "the attribute named *data_type* can hold either the value *OceanSITES metadata,* or *OceanSITES profile data***ò**; "the dimension of the variable named *TIME* is the dimension *TIME***ò**. Thus, *CdmCL* is structured by three abstract classes DimensionConstraint, VariableConstraint and AttributeConstraint (see figure 3). Then, to express a constraint related to a specific instance, *CdmCL* follows the three different scenarios, defining nine concrete concepts:

- Constraint is applied to a specific CDM concept referenced by its *name*: *aDimensionConstraint*, *aVariableConstraint* or *anAttributeConstraint.*

- Constraint is applied to items whose *name* matches a specific regular expression: *TemplateDimensionRegex*, *TemplateVariableRegex*, or *TemplateAttributeRegex*.

- Constraint is applied to a set of items, characterized by a set of names: *TemplateDimensionList*, *TemplateVariableList* or *TemplateAttributeList*.

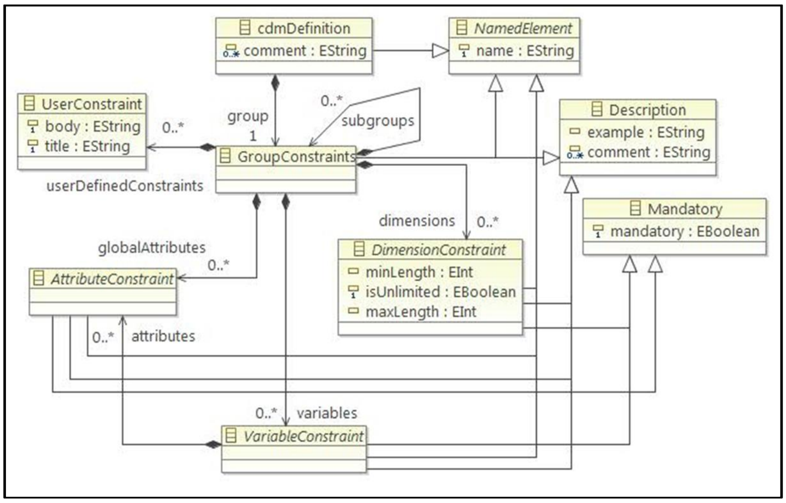


Fig. 3. An excerpt of the ecore *CdmCL* model (CdmCL.ecore)

4.2 Common Constraints

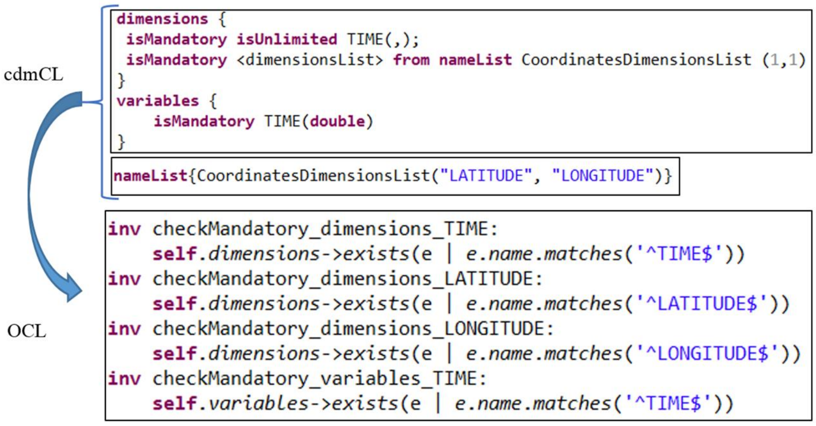
The following section introduces the constraints that are common between *DimensionConstraint*, *VariableConstraint* and *AttributeConstraint*.

Mandatory: This constraint verifies that the *name* of the related concept exists. For the template list concept, an extra *Boolean* attribute *or* permits to indicate whether one of the items of the list is mandatory or all the items are mandatory. As an example, figure 4 presents the *CdmCL* expressions and the corresponding OCL statements for *aDimension* called *DEPTH* (a single dimension) and a *TemplateDimensionList* (coordinate dimension list holding *LATITUDE* and *LONGITUDE*).
The values between parentheses permit the definition of dimension length and are explained in the next section. Figure 5 presents a CDM instance to the left respecting the OceanSITES standard, thus the instance is valid, whereas the other instance is not valid because the mandatory dimension *LONGITUDE* and the mandatory variable *TIME* are missing, thus the OCL constraints checkMandatory_dimension_LONGITUDE and checkMandatory_variables_TIME are violated.

Repetition: This constraint checks that a *name* of a concept is never repeated.

Format: This constraint verifies that names of a set of concept have a specific format, either uppercase, lowercase or matches a specific regular expression.

*User Defined Constraints***:** This concept increases the flexibility of the language, by allowing the user to enter manually an OCL statement. However, it is the user's responsibility to verify the correctness of the OCL statement regarding the CDM metamodel. These constraints are defined in the context of *Group*.



```
dimensions {
 isMandatory isUnlimited TIME(,);
 isMandatory <dimensionsList> from nameList CoordinatesDimensionsList (1,1)
}
variables {
    isMandatory TIME(double)
}
```

```
nameList{CoordinatesDimensionsList("LATITUDE", "LONGITUDE")}
```

```
inv checkMandatory_dimensions_TIME:
    self.dimensions->exists(e | e.name.matches('^TIME$'))
inv checkMandatory_dimensions_LATITUDE:
    self.dimensions->exists(e | e.name.matches('^LATITUDE$'))
inv checkMandatory_dimensions_LONGITUDE:
    self.dimensions->exists(e | e.name.matches('^LONGITUDE$'))
inv checkMandatory_variables_TIME:
    self.variables->exists(e | e.name.matches('^TIME$'))
```

Fig. 4. Mandatory constraint example with *CdmCL* and OCL correspondence

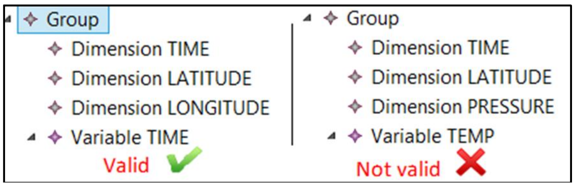


Fig. 5. CDM data respecting and not respecting OceanSITES standards

4.3 Concept Related Constraints

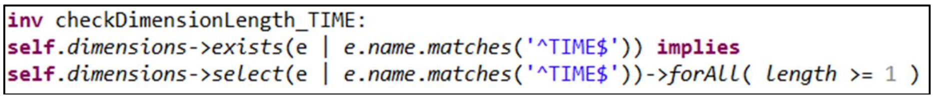
In addition, other constraints are related to specific concepts.

- **Dimensions length constraints**: The *length* value for a dimension can be a limited or unlimited (any positive value). For the limited length, it can be a specific value, a value in a range, greater than or equal to a specific value, smaller than or equal. To achieve these constraint objectives, *dimensionConstraint* concept has two Integer attributes called *minLength* and *maxLength* and one Boolean attribute called *IsUnlimited* (see table 1).

case	*Min*	*Max*	*isUnlimited*	*Description*
1	-	-	true	Length is unlimited, i.e. any positive value
2	x	x	false	Length is equal to x, a specific value
3	x	y	false	Length is between x and y, y>x>0
4	x	-1	false	Length is greater than x
5	-1	y	false	Length is smaller than y
6	-1	-1	false	No constraint on length

Table 1. Dimension *length* constraints categories

Figure 6 illustrates the OCL statement generated for *aDimension TIME* with *unlimited* dimension length who's *CdmCL* is given in figure 4. The concept related constraints are built all in the same way. The first part of the OCL constraint (exists, select) defines the context of the specific concept (here the TIME Dimension). Then, the second part expresses the constraint itself (forAll).



```
inv checkDimensionLength_TIME:
self.dimensions->exists(e | e.name.matches('^TIME$')) implies
self.dimensions->select(e | e.name.matches('^TIME$'))->forAll( length >= 1 )
```

Fig. 6. Dimension length OCL constraint

- **Variable shape constraint**: A variable is characterized by a set of shapes i.e. a set of dimensions (see figure 1). A variable can have a shape of the same name as the variable's name. For example a variable named *TIME* is associated with a dimension named *TIME*. Moreover, a variable can be associated with a dimension or a set of dimensions. For example, a variable named *TIME_QC* is associated with dimension named *TIME*. To accomplish this type of verification, we have two concepts *SimilarDimensionConstraint* and/or a set of *PredefinedShape* concept for a *VariableConstraint.* On one hand *SimilarDimensionConstraint* concept allow the generation of an OCL invariant that verifies that a variable is associated with a dimension of same variable's name. On the other hand *PredefinedShape* concept permits to verify that a variable is associated with any preexisting dimensions. Figure 7 illustrates the previous discussion and presents the *CdmCL* representation along with the OCL to be generated.
- **VariableConstraint and AttributeConstraint DataType**: This constraint verifies that the variables and the attributes can have any data type from a list of data types.
- **Attribute Value Constraint**: A constraint on the *value* of an *Attribute*, the possible categories are given in table 2.

Case	Constraint
Unique value	The value should have this and only this value
Regular Expr	The value should match the regular expression
Range	A range of values between min and max, similar to dimension length
List	The value can be any value from a list of values.
Standard	The value matches a regex given by a standard (e.g. ISO8601).

Table 2. Attribute's value constraints categories

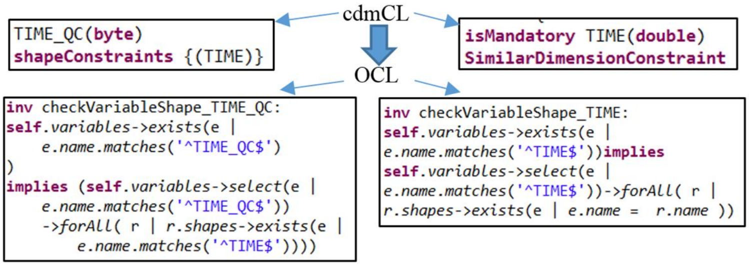
5 Conformity tool and experiment

Based on *CdmCL*, we developed a conformity checker for netCDF[1] files which conform to CDM metamodel. The tool architecture is shown in figure 8. It is developed in Java and based on the Eclipse Modeling Framework (EMF). First, the tool transforms the *CdmCL* model into OCL statements in a separated file. This part

---

[1] NetCDF (Network Common Data Form) is a CDM compliant file serialization format

(*CdmCL2OCL*) uses Xtext API, and the library we developed for the transformation of *CdmCL* expressions to OCL constraints. Then, the tool transforms a NetCDF file into an instance of CDM model. This part (*nc2CDM*) is simply based on the Java NetCDF API and on the CDM Java API (provided by EMF). Finally, the tool checks the conformity of the CDM file to the CDM metamodel enriched with the generated OCL file, and indicates whether it is valid or not. This part is based on the integrated OCL checker of the EMF modeling tool.



```
TIME_QC(byte)
shapeConstraints {(TIME)}
```

```
isMandatory TIME(double)
SimilarDimensionConstraint
```

```
inv checkVariableShape_TIME_QC:
self.variables->exists(e |
    e.name.matches('^TIME_QC$')
)
implies (self.variables->select(e |
    e.name.matches('^TIME_QC$'))
    ->forAll( r | r.shapes->exists(e |
        e.name.matches('^TIME$'))))
```

```
inv checkVariableShape_TIME:
self.variables->exists(e |
e.name.matches('^TIME$'))implies
self.variables->select(e |
e.name.matches('^TIME$'))->forAll( r |
r.shapes->exists(e | e.name =  r.name ))
```

Fig. 7. CdmCL representation for two variables one with *PredefinedShape* concept and the other with *SimilarDimensionConstraint* concept and their corresponding OCL

The tool has been tested on OceanSITES. Due to the lack of space we express only the OceanSITES global attributes standard given in figure 2 in *CdmCL*. The *CdmCL* shown in figure 9 indicates that a global attribute named *data_type* is mandatory, of type *string* and has any of the values presented by the list *dataTypeGlobalList*. This list hold the values *OceanSITES metadata*, *OceanSITES profile data*, *OceanSITES time-series data* or *OceanSITES trajectory data*. Furthermore, it indicates that a global attribute named *format_version*, of type *string* and should hold the value "*1.1*". The *CdmCL* expressions and the generated OCL after using CdmCL2OCL tool are given in figure 9.

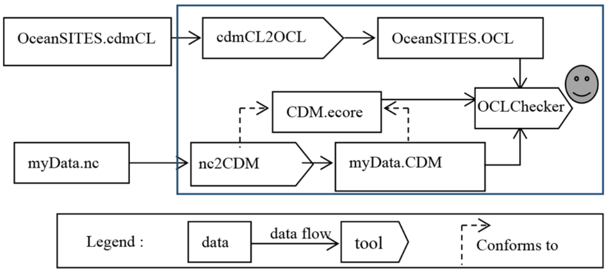
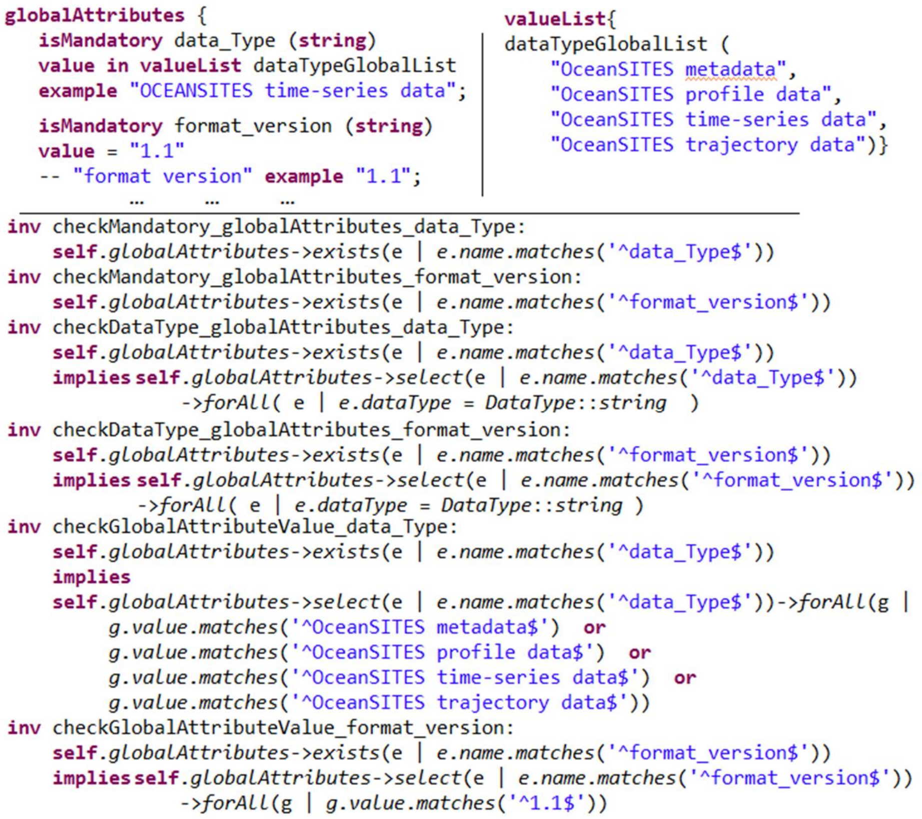


Fig. 8. *CdmCL* conformity tool architecture

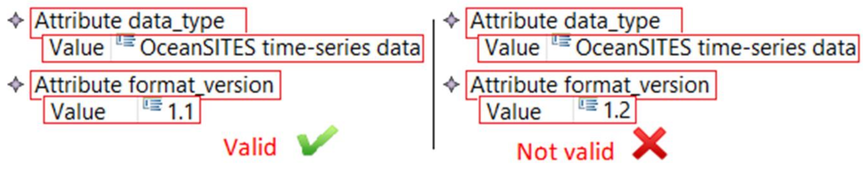
The generated OCL file is then used with the CDM model to check the conformity of an input netCDF file. If the input file does not respect the constraints, a message indicates that the input file does not conform to CDM model with the set of OCL violated constraints. Figure 10 presents an excerpt of CDM data converted from a netcdf file. These data are validated with this generated OCL. It is seen that the left instance is valid whereas the right instance is invalid because the global attribute *format_version* value constraint is violated.



```
globalAttributes {                               valueList{
   isMandatory data_Type (string)             |  dataTypeGlobalList (
   value in valueList dataTypeGlobalList      |      "OceanSITES metadata",
   example "OCEANSITES time-series data";     |      "OceanSITES profile data",
                                              |      "OceanSITES time-series data",
   isMandatory format_version (string)        |      "OceanSITES trajectory data")}
   value = "1.1"                              |
   -- "format version" example "1.1";         |
        ...     ...     ...
inv checkMandatory_globalAttributes_data_Type:
    self.globalAttributes->exists(e | e.name.matches('^data_Type$'))
inv checkMandatory_globalAttributes_format_version:
    self.globalAttributes->exists(e | e.name.matches('^format_version$'))
inv checkDataType_globalAttributes_data_Type:
    self.globalAttributes->exists(e | e.name.matches('^data_Type$'))
    implies self.globalAttributes->select(e | e.name.matches('^data_Type$'))
              ->forAll( e | e.dataType = DataType::string  )
inv checkDataType_globalAttributes_format_version:
    self.globalAttributes->exists(e | e.name.matches('^format_version$'))
    implies self.globalAttributes->select(e | e.name.matches('^format_version$'))
           ->forAll( e | e.dataType = DataType::string )
inv checkGlobalAttributeValue_data_Type:
    self.globalAttributes->exists(e | e.name.matches('^data_Type$'))
    implies
    self.globalAttributes->select(e | e.name.matches('^data_Type$'))->forAll(g |
         g.value.matches('^OceanSITES metadata$')  or
         g.value.matches('^OceanSITES profile data$')  or
         g.value.matches('^OceanSITES time-series data$')  or
         g.value.matches('^OceanSITES trajectory data$'))
inv checkGlobalAttributeValue_format_version:
    self.globalAttributes->exists(e | e.name.matches('^format_version$'))
    implies self.globalAttributes->select(e | e.name.matches('^format_version$'))
              ->forAll(g | g.value.matches('^1.1$'))
```

Fig. 9. *CdmCL* and generated OCL for OceanSITES

Figure 10 illustrates that the right version is invalid because *format_version* constraints was violated, but in reality this global attribute can have the value of 1.2 and even 1.3 since OceanSITES standard has been evaluated to the new version 1.3 with backward compatibility. Therefore by just modifying the CdmCL *format_version* value to hold the values (1.1, 1.2 or 1.3), we can have the new standard OCL representation without any professional interference and the instance will be valid with respect to OceanSITES version 1.3.

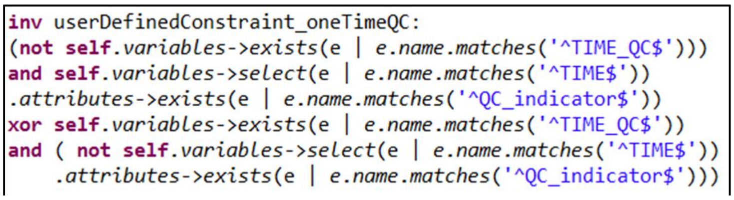


Attribute data_type
Value OceanSITES time-series data
Attribute format_version
Value 1.1

Attribute data_type
Value OceanSITES time-series data
Attribute format_version
Value 1.2

Fig. 10. Valid and Invalid cdm instance with respect to OceanSITES standard version 1.1

Experimenting *CdmCL* on OceanSITES, it is observed that more than 90% percent of the constraints are achieved in the *CdmCL* except the constraints that are related to multiple CDM concepts at the same time. For example, a constraint verifies the existence of either a variable named *TIME* with attribute named *QC_indicator* or a variable named *TIME_QC*.



```
inv userDefinedConstraint_oneTimeQC:
(not self.variables->exists(e | e.name.matches('^TIME_QC$')))
and self.variables->select(e | e.name.matches('^TIME$'))
.attributes->exists(e | e.name.matches('^QC_indicator$'))
xor self.variables->exists(e | e.name.matches('^TIME_QC$'))
and ( not self.variables->select(e | e.name.matches('^TIME$'))
    .attributes->exists(e | e.name.matches('^QC_indicator$')))
```

Fig. 11. User defined OCL constraints

For *CdmCL* syntax readability, the two global attributes of OceanSITES given in figure 2 are represented by approximately eight lines in *CdmCL* expressions and generate around 30 lines of OCL statements. One page of textual constraints from the standard is expressed by around 25 lines in *CdmCL* and generates around 300 lines of OCL.

The language CdmCL was introduced to OceanSITES users (standard reader) and they confirm the expressiveness and the readability of *CdmCL*.

6 Related Works

Several studies are proposed to support OCL integration on modeling processes. The Dresden OCL Toolkit [9] proposes an OCL library and the recent version provides an OCL-to-Java Code Generator. USE (UML-based Specification Environment) [10] is a tool to facilitate the validation of UML models with OCL constraints. USE supports consistency, independence of constraints, and relevance of constraints analysis. These OCL tools are complementary to the proposed approach and can be used to facilitate the management of the generated OCL constraints.

In the domain of OCL generation, [11] propose OCL automatic generation from UML class diagrams. The approach aims at simplifying the process of generation of OCL statements. The approach involves expressing constraints by a class diagram. In the addressed domain, most of the constraints are related to specific instances. Following this approach would result in too many classes (one per instance constrained), and the class diagram syntax is far from the scientific standard edition.

In [12], the authors propose to convert natural language expressions to the equivalent OCL statements. The expressions are constraints and pre/post conditions related to UML diagrams. OCL generation is based on the Semantic Business Vocabulary and Rules language (SBVR) to avoid inconsistencies. With *CdmCL*, we prefer to define a DSL because, in a small and well identified domain, it promotes the development of efficient and accurate solutions [13]. As a DSL, CdmCL disambiguates scientific standard edition. And in addition, it is close enough to the natural language so that its *use* does not require technical skills on OCL. However, the *creation* of a DSL usually requires both domain knowledge and language development expertise [13]. But, the production cost of *CdmCL* is low since it relies directly on OCL semantics, while hiding unnecessary OCL features. Thus, translation to OCL may be directly done without considering SBVR intermediate level.

7 Conclusion

This paper proposes a domain specific constraint language for CDM. The language structure is based on CDM structure and results in a set of constraint categories. These categories permit to define constraints in a human readable language and serves in the automatic generation of OCL. The approach hides the complexity of writing OCL manually and increases the productivity by generating a large number of OCL statements for few lines written in *CdmCL*. Furthermore, any standard edition and conformity checking can be done easily without OCL and programming interference.

In perspective, we are working on developing domain specific operators to automate scientific data migration from a standard to another one.

8 References

1. *Unidata, Common Data Model (CDM).* Version 4. http://www.unidata.ucar.edu/software/thredds/current/netcdf-java/CDM/
2. *OceanSITES Userôs Manual*, Version 1.2. http://www.oceansites.org/
3. Object Management Group: *Object Constraint Language (OCL) Specification*, Version, 2.4, http://www.omg.org/spec/OCL/2.3.1/
4. Meyer B., *Object-Oriented Software Construction,* International Series in Computer Science, Second Edition, Prentice-Hall (1997)
5. Linehan M., *Ontologies and rules in Business Models*, in the 11h IEEE Enterprise Distributed Object Conference (EDOC), 149-156, 2008.
6. Wahler M., *Using Patterns to Develop Consistent Design Constraints*, PhD Thesis, ETH Zurich, Switzerland, 2008.
7. *Eclipse Modeling Framework* (EMF). http://www.eclipse.org/modeling/emf/
8. *OceanSITES file format checker*. http://www.coriolis.eu.org/Observing-the-ocean/Observing-system-networks/OceanSITES/Access-to-data.
9. Schütze L., Wilke C., Demut B., *Tool-Supported Step-By-Step Debugging for the Object Constraint Language,* In OCL@MODELS 2013, 2013.
10. Gogolla M., et al. *USE: A UML-Based Specification Environment for Validating UML and OCL*, Science of Computer Programming, Vol. 69, 27-34, 2007.
11. Li Tan, Zongyuan Yang and Jinkui Xie, *OCL Constraints Automatic Generation for UML Class Diagram*, IEEE International Conference on Software Engineering and Service Sciences (ICSESS) 392 – 395, 2010.
12. Bajwa I.S., Bordbar B. and Lee M.G., *OCL Constraints Generation from Natural Language Specification*, in the 14th IEEE Enterprise Distributed Object Conference (EDOC), 204-213, 2010.
13. Mernik M., Heering J.and Sloane A. *When and how to develop domain-specific languages* in ACM Computing Surveys, 37(4), 316-344, 2005.

Towards Fixing Sketchy UML Models by Leveraging Textual Notations: Application to Real-Time Embedded Systems

Frédéric Jouault and Jérôme Delatour

LUNAM, L'Université Nantes Angers Le Mans
TRAME team, ESEO, Angers, France
`firstname.lastname@eseo.fr`

Abstract. During the development of real-time embedded system, the use of UML models as blueprints is a common practice with a focus on understandability rather than comprehensiveness. However, further in the development, these models must be completed in order to achieve the necessary validation and verification activities. They are typically performed by using several formal tools. Each tool needs models of the system at a given abstraction level, which are precise and complete enough with respect to how the tool processes them. If UML is appropriate for capturing multiple concerns, its multiple partial views without a global one increase the difficulty of locating inconsistency or incompleteness issues. Therefore, ensuring completeness is time consuming, fastidious, and error prone. We propose an approach based on the use of a UML textual syntax closely aligned to its metamodel: tUML. An initial prototype is described, and examples are given.

1 Introduction

Using sketchy models is common during software systems development. Such models (also called contemplative models) are just drawings that cannot be automatically processed (e.g., transformed into code, or verified). In the use of such models as blueprints, the focus is on communication rather than on completeness. The UML language is a good candidate for that kind of practice. It is primarily a graphical notation, and it offers a relatively large set of diagrams for representing various concerns. Moreover, as it is a standard, one can expect that it is known by developers, and also by stakeholders. Therefore, it helps communicating a better understanding about the system under study, and supports a better collaboration with various specialists.

However, such contemplative use of UML models is only partially satisfactory. Part of the effort to produce them cannot be automatically reused in further development phases. Going from contemplative to productive models is a key challenge of model-driven approaches [4]. In these approaches, development can be seen as a set of successive model transformations steps in order to produce code from analysis models. Each step should be partially automated.

In real-time embedded system development, this is even more important. The reliability and complexity of such systems require numerous verifications and validations activities all along the development process. These activities are based on the use of tools such as model checkers, theorem provers, code generators, and test generators. The use of UML in this context presents some issues. It has no standard formal semantics, and many semantic variation points. A many-diagram model may contain inconsistencies, and no global view helps their discovery. UML is semi-formal: a given drawing may have different interpretations. UML expressivity is not completely satisfactory for this domain.

Solutions to these issues have been developed. UML extensions have been defined (e.g., the MARTE[1] profile) to adapt it to the real-time domain. Numerous works propose a variety of translations from part of UML to different formal languages [6,10,9]. These translations enable detection of some inconsistencies [5], and other verification activities.

These solutions require that the verified model parts be precise, complete, and unambiguous. Other parts may remain sketchy [12]. However, it is difficult for an engineer to go from a sketchy model to a precise one. This typically requires detailed knowledge of: the UML specification, and the specific semantics given to it by a given verification tool [2]. Even if some UML editors assist users, this work is difficult and long. It is generally necessary to fill-in hundreds of property sheets accessible from a potentially large number of distinct graphical views.

In this work, we propose an approach to bridge the gap between sketchy and (partially) precise models. We put some efforts in making it as independent of specific UML editors as possible. It is based on the use of a specific textual syntax called tUML that closely follows the UML metamodel structure, as well as on the use of additional constraints on models. Neither text nor graphical is better than the other. However, by complementing graphical tools with textual ones, we can get both their respective advantages.

We illustrate our approach with the pacemaker case study presented in [3]. It is actually while working on this case study that we initially had the idea to develop the presented approach. We started by playing the role of the designer defining a sketchy model. Then, we had to complete it so that it can be verified. This task is now much easier thanks to our tUML prototype.

Section 2 presents the approach in four steps: an overview of tUML (Section 2.1), a description of our prototype (Section 2.2), an explanation of its usage (Section 2.3), and a discussion (Section 2.4). Then, Section 3 compares tUML to some related works. Finally, we conclude in Section 4.

2 Approach

2.1 tUML Presentation

tUML is a textual concrete syntax for a subset of the standard UML metamodel. A detailed definition and justification of this subset is beyond the scope of this

---

[1] `http://www.omgmarte.org/`

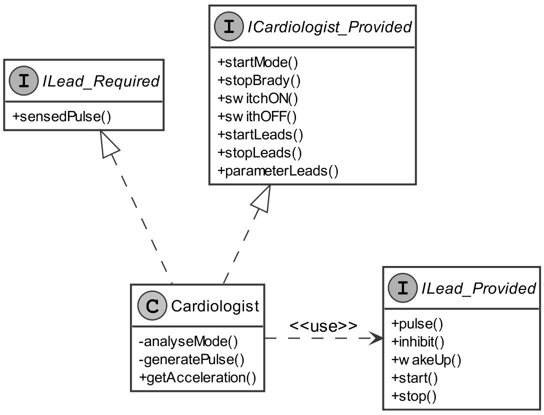


Fig. 1. Class diagram of `Cardiologist`

paper. Roughly, it consists of the parts of the UML metamodel that correspond to the three following diagrams: 1) class diagrams, 2) composite structure diagrams, and 3) state diagram. These diagrams are typically the ones used to model real-time embedded systems. We do not attempt to cover the whole UML metamodel, but built our subset bottom-up, making sure we can verify what we model [8]. Additionally, we use a specific action language restricted to what we can verify.

In order to illustrate: 1) the tUML textual syntax, and 2) its relations with the standard graphical notation, we use the pacemaker case study introduced in [3]. In this case study, a pacemaker is designed as a UML model, of which a full description does not belong here. Therefore, we only briefly mention what is necessary to understand this paper. A `Cardiologist` active object is responsible for applying a heart-assistance policy such as `TriggeredPacing` by piloting `Leads`. A `Lead` active object brokers access to actual physical leads placed on the heart of the patient. `Cardiologist` and `Lead` are software components that are part of an implanted `PulseGenerator`. The overall `System` consists of a `PulseGenerator`, which can communicate with an external `DeviceControlMonitor`. We reuse three figures from [3], which correspond to our three UML metamodel parts of interest: classes, composite structure, and state machines.

Classes. Figure 1 (corresponding to Figure 9 from [3]) is a partial view of class `Cardiologist`, and related interfaces in the standard class diagram notation. `Cardiologist` implements interface `ICardiologist_Provided`[2], which defines messages it can receive from its controller. It also implements interface `ILead_Required`, which defines messages it can receive from `Leads`. `Cardiologist` uses interface `ILead_Provided`, which defines messages it can send to `Leads`.

Listing 1 is an excerpt of the corresponding tUML definition. Class `Cardiologist` is defined as implementing two interfaces (line 1), and using a third one

---

[2] All interface names from original case study have been simplified to increase clarity.

Listing 1. `Cardiologist` excerpt

```
1  class Cardiologist behavesAs SMCardiologist implements
     ↪ICardiologist_Provided, ILead_Required {
2   // [ports]
3   private operation analyseMode();
4   private operation generatePulse();
5   public operation getAcceleration();
6   stateMachine SMCardiologist {
7     region Region0 {
8       // [state machine contents]
9  }}}
10 usage Usage0 of ILead_Provided by Cardiologist;
```

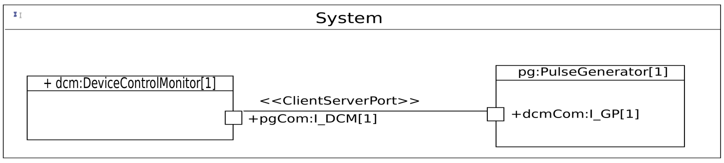


Fig. 2. Composite Structure diagram of `System`

(line 10). These interfaces have the same names as those in Figure 1. Lines 3 to 5 contain the definitions of operations owned by the class. Comments starting with two slashes end with the line. They are used here to stand for actual ports (line 2) and state machine contents (line 8), which are not detailed. Lines 6 to 9 correspond to an element not shown in the class diagram: a state machine (not detailed here) that specifies the behavior of this active object.

Composite Structure. Figure 2 (Figure 5 in [3]) is a partial view of the `System`'s architecture in the standard composite structure diagram notation. This architecture consists of a `DeviceControlMonitor` named `dcm` owning a port called `pgCom`, a `PulseGenerator` named `pg` owning a port called `dcmCom`, and a connector between the ports.

Listing 2 is an excerpt of the corresponding tUML definition. Class `System` is defined at lines 1-4 as consisting of a `PulseGenerator` part at line 2, a `DeviceControlMonitor` part at line 3, and a connector at line 4. Class `PulseGenerator` is defined at lines 5-9 with a port at line 7. This class is also composite but its parts (line 6) and connectors (line 8) are not detailed here. Class `DeviceControlMonitor` is defined at lines 10-11, and only contains a port (line 11).

State Machines. Figure 3 (Figure 12 in [3]) is a partial view of a composite substate of `Cardiologist`'s behavioral state machine in the standard state diagram notation. State `TriggeredPacing` is composed of states `StartingPulse` and `WaitingPulse`, as well as of an initial pseudo-state. It also contains four transitions: 1) one transition from the initial pseudo-state to `StartingPulse`, 2) one transition from `StartingPulse` to `WaitingPulse` triggered after a given

Listing 2. `System` composite structure

```
class System {
  public composite pg[1-1] unique : PulseGenerator;
  public composite dcm[1-1] unique : DeviceControlMonitor;
  connector Connector0 between dcm.pgCom and pg.dcmCom; }
class PulseGenerator {
  // [parts]
  public composite port dcmCom[1-1];
  // [connectors between parts]
}
class DeviceControlMonitor {
  public composite port pgCom[1-1]; }
```

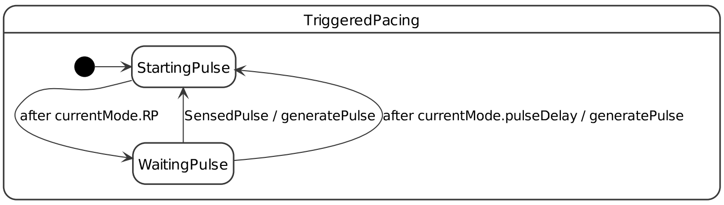

TriggeredPacing
StartingPulse
after currentMode.RP
SensedPulse / generatePulse
after currentMode.pulseDelay / generatePulse
WaitingPulse

Fig. 3. State diagram showing part of the behavior of `Cardiologist`

time (`currentMode.RP`), and two transitions from `WaitingPulse` to `StartingPulse` triggered: 3) by `SensedPulse` message, with `generatePulse` effect, and 4) after a given time (`currentMode.pulseDelay`), with the same effect.

Listing 3 is an excerpt of the corresponding tUML definition. State `TriggeredPacing` is defined at lines 1-9 as consisting of region `Region0`. This region itself consist of an initial pseudo-state (line 7), two states `StartingPulse` (lines 8) & `WaitingPulse` (line 9), and four transitions (lines 3-6). Each transition has a name. Each transition with a trigger refers to an event by name (escaped between double quotes because they contain spaces). Events are defined in a part of the model not shown here. Finally, the two transitions having an effect specify it as an empty opaque behavior (lines 5-6), with only its name specified.

2.2 Prototype

Now that tUML has been presented, we will describe its current prototype implementation. Eclipse Modeling[3] has been used in the following way:

- **Eclipse UML** provides the UML abstract syntax implementation (very close to the standard), on top of EMF (Eclipse Modeling Framework). Tools based on Eclipse UML (e.g., Papyrus) are directly compatible with tUML.
- **TCS** [7] is used to define the textual syntax. It provides three elements: 1) an *extractor* from Eclipse UML abstract syntax to concrete tUML textual syntax, 2) an *injector* from tUML textual syntax to Eclipse UML, and 3) an *editor* (see Figure 4) tuned for tUML textual syntax.

---

[3] `http://www.eclipse.org/modeling/`

Listing 3. `Cardiologist` state machine except

```
state TriggeredPacing {
  region Region0 {
    Initial0->StartingPulse : Transition0;
    StartingPulse->WaitingPulse : Transition1 : "TE - currentMode.RP" /;
    WaitingPulse->StartingPulse : Transition2 : "SE - SensedPulse" /
        ↪opaqueBehavior generatePulse;;
    WaitingPulse->StartingPulse : Transition3 : "TE - currentMode.
        ↪pulseDelay" / opaqueBehavior generatePulse;;
    initial pseudoState Initial0;
    state StartingPulse;
    state WaitingPulse;   }}
```

- **ATL** [1] transformations are used to implement four elements: 1) *constraints* (in OCL) to check on tUML models, 2) *qualified names* computation and resolution whereas TCS only supports simple names by itself, 3) *visualization* in the form of a PlantUML[4] export, and 4) *bridges* with other tools.

Other tools could have been used (e.g., Xtext), but we prefer familiar ones.

The PlantUML export enables graphical rendering of tUML models. Figures 1 and 3 have actually been generated through this process. Figure 2 is however still the original from [3] because PlantUML cannot properly render it yet. A non-standard context diagram can also be rendered. It is a specialization of the communication diagram in which a non-ordered list of all exchanged messages (with directions) is shown on each edge.

Our prototype notably has the following limitations. Conversions between textual and abstract syntax currently work on whole models, and graphical information is not synchronized yet. Fine-grain synchronization would be helpful but is not mandatory in our approach. Apart from tools based on Eclipse UML, only a Rhapsody import has been implemented so far.

2.3 Using tUML to Fix Sketchy Models

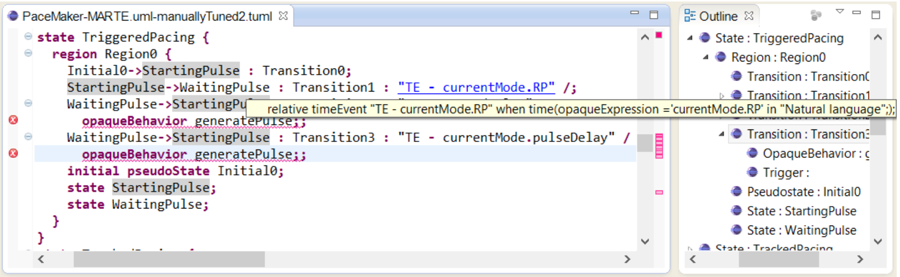
The previous sections gave an overview of tUML and of its prototype implementation. This section will give an overview of how they can be used to fix sketchy models. Before creating models and fixing them, it is first necessary to consider how they will be used (e.g., for code generation, or verification). Then, an expert can define what a *correct* model precisely is in that context. This will typically lead to a list of custom validation constraints. This work does not need to be performed for each model, but only for each usage of models (e.g., for a given code generator, or for a given verification tool).

The user typically follows the five-step process described below:

1. **Creating Sketchy Model.** A modeling tool is first used to create a sketchy model. This is generally done with a graphical UML editor.
2. **Converting to tUML.** Using the extraction tool presented previously, the user serializes the model in tUML.

---

[4] `http://plantuml.com/`



```
state TriggeredPacing {
  region Region0 {
    Initial0->StartingPulse : Transition0;
    StartingPulse->WaitingPulse : Transition1 : "TE - currentMode.RP" /;
    WaitingPulse->StartingPu
      opaqueBehavior generatePulse;;
    WaitingPulse->StartingPulse : Transition3 : "TE - currentMode.pulseDelay" /
      opaqueBehavior generatePulse;;
    initial pseudoState Initial0;
    state StartingPulse;
    state WaitingPulse;
  }
}
```

```
relative timeEvent "TE - currentMode.RP" when time(opaqueExpression ='currentMode.RP' in "Natural language";);
```

Fig. 4. Screenshot of tUML editor showing composite state `TriggeredPacing`

3. **Identifying Problems.** Issues are identified by applying standard UML constraints, as well as custom constraints if applicable. They are shown in the tUML editor. If no problem is identified, then we can go to step 5
4. **Fixing Problems.** The user can tune the model in order to fix problems. This is often easier to do textually because all information is presented at once. In comparison, a graphical editor often requires opening various property sheets. Then, we go back to step 3 in order to check if the tackled issues are fixed, and if some issues remain.
5. **Use Fixed Model.** Now that all identified problems have been fixed, the model can be used in four notable ways: 1) **reloading** the model in the original UML tool, or another one, 2) **generating** code, 3) **verifying** the model using formal methods, and 4) **rendering** the model using tools such as PlantUML (see prototype description).

Our pacemaker example was initially created as a sketchy model, not to be automatically processed. Later, we decided to attempt to use some verification tools [8]. We defined five specific constraints in addition to the standard ones. tUML helped us discover and pinpoint (see error markers in Figure 4) more than thirty issues mostly in the modeling of communication. We can notably mention: 1) opaque behaviors with no body (e.g., in Listing 3, which is not obvious on Figure 3), 2) undetailed communications (e.g., call to private operation `generatePulse` with no body), and 3) use of non-defined signals in triggers. To fix them, we select a language for opaque behaviors, and we fill-in the blanks.

2.4 Discussion

After presenting tUML, its implementation, and its usage, we now discuss the advantages and drawbacks of our approach. Firstly, tUML has all the benefits of a textual language, such as: 1) global search and replace, 2) relatively easy editing with no need to click through numerous property pages, 3) support for diff/patch and text-based version control systems, and 4) no need to focus on graphical layout (indentation being much easier). Other advantages of tUML when compared with the standard UML graphical notation include:

- **Transparency.** All model elements are immediately visible, and no aspect of the model is hidden away in property sheets accessible via multiple clicks.
- **Global view.** Whereas the graphical notation fragments the model into separate diagrams, tUML provides a single global view.
- **Reduced redundancy.** There is no need to duplicate elements (even partially) across various diagrams.
- **Proximity to metamodel.** The textual syntax of tUML follows the UML metamodel very closely.

Proximity to the UML metamodel notably has the advantage that it is relatively easy to figure out where a given aspect of the UML metamodel can be accessed. Moreover, this enables relatively precise reporting of errors as text markers, even though they are actually identified on the abstract syntax. Graphical editors cannot always do this. For instance, in Papyrus, some errors on class members are shown on their owning class in the class diagram.

Although this is not due to the textual nature of tUML, the automatic generation of views presents some advantages as well. Even though tUML models typically come from graphical UML editors, this has the advantage of reformulating the models in a different way, possibly helping to spot issues. Moreover, the non-standard context diagram is especially helpful in finding messages that are missing (e.g., when their emission or reception are not properly modeled).

Of course, tUML also has some limitations:

- **Verbose.** All model elements have to appear in a single text file. Therefore, even though they are visible, details of interest may actually be drowned in unnecessary details. Global view is thus both an advantage and a drawback. To mitigate this issue, generation of partial textual views should be investigated. Synchronization with the whole model may become an issue.
- **Non-standard.** Because it is not standard, tUML needs to be learned even by UML experts. This adds one difficulty to its usage.
- **Incomplete.** Only a subset of UML is currently supported. This subset is especially targeted at real-time embedded systems. Using tUML in other contexts may prove difficult (e.g., if one needs activities).
- **Textual.** Finally, even though most advantages of tUML are due to its textual nature, some users may not accept to type UML code.

3 Related Work

UML is best known for its graphical notation, but tUML is not the first proposed UML textual notation. The OMG has notably standardized two of its ancestors[5]:

- **HUTN** initially seems to provide a solution since it promises automatic derivation of a human-usable textual notation from any metamodel. However, this genericity leads to relatively verbose syntaxes.

---

[5] `http://www.omg.org/spec/HUTN/`, and `http://www.omg.org/spec/ALF/`

- **Alf** defines a specific textual syntax for a UML subset. The OMG thus acknowledges the fact that HUTN is not fully adapted to this kind of use. Unfortunately, Alf is limited to a different subset of UML, which is not adapted to real-time embedded systems. tUML is thus significantly different from Alf. Notably, Alf only provides activities to specify behavior, whereas tUML focuses on state machines. Moreover, Alf specifies a new metamodel that is closer to its concrete syntax than the standard UML metamodel. Although a bidirectional mapping to the standard UML metamodel is also provided, such a gap between textual and graphical notations may make the job of fixing models for formal verification purposes more difficult.

Most non-standard textual UML notations aim at producing sketchy models by leveraging auto-layout tools. We cannot describe them all[6] here, but two notable ones are:

- **TextUML**[7] is relatively close to tUML but does not follow the UML metamodel as closely (notably wrt. its action language).
- **PlantUML** has the advantage of being less verbose and more permissive than all other mentioned approaches. However, its objective is only to automatically generate visual diagrams. It does not follow the standard metamodel. Moreover, it is so permissive that many elements (e.g., class members, and transition labels) can be represented by arbitrary text. Apart from following very strict conventions, there is no way to guarantee that these elements actually follow the UML syntax.

Some approaches like UML/P [11] also offer a textual representation of UML models, and specific constraints to check them. However, the supported UML subset differs: UML/P is not specific to real-time embedded systems, and notably includes additional diagrams that our approach [8] is not yet able to verify. It also seems to lack composite structure diagram, which we need.

4 Conclusion

In this paper, we have presented an approach that helps fixing incomplete UML models to make them suitable as input for verification and validation activities and tools. This approach relies on tUML: a specific textual notation for UML that notably follows its metamodel very closely. A prototype implementation has been used on a couple of case studies (including the pacemaker one presented here), and half a dozen different models. tUML significantly helped in discovering issues, and therefore in fixing them.

We have so far demonstrated the worth of the approach, but its development is still in progress. It should be further studied and refined, notably by applying it to more case studies, including industrial ones. This would lead to a complete evaluation. This should bring out more kinds of constraints to check on models,

---

[6] See `http://modeling-languages.com/uml-tools/#textual` for a bigger list.

[7] `https://github.com/abstratt/textuml/`

as well as help figuring out in which direction the supported UML subset should be extended. All sketchy models issues cannot be resolved automatically. It could nonetheless be useful to provide quick fixes[8] in order to not only pinpoint issues, but also partially automate their resolution.

A possible extensions of our work on tUML would be to propose it to the OMG as an extension of Alf, which notably lacks support for state machines.

References

1. Bézivin, J., Jouault, F.: Using ATL for Checking Models. Electronic Notes in Theoretical Computer Science 152, 69–81 (Mar 2006), proceedings of the International Workshop on Graph and Model Transformation (GraMoT 2005)
2. Broy, M., Cengarle, M.: UML formal semantics: lessons learned. Software & Systems Modeling 10(4), 441–446 (2011)
3. Delatour, J., Champeau, J.: Embedded Systems: Analysis and Modeling with SysML, UML and AADL, chap. Case Study Modeling using MARTE, pp. 139–156. Wiley-ISTE, West Sussex, England (Apr 2013)
4. Gérard, S., Feiler, P., Rolland, J.F., Filali, M., Reiser, M.O., Delanote, D., Berbers, Y., Pautet, L., Perseil, I.: UML & AADL '2007 Grand Challenges. SIGBED Rev. 4(4) (Oct 2007)
5. Gogolla, M., Kuhlmann, M., Hamann, L.: Consistency, Independence and Consequences in UML and OCL Models. In: Dubois, C. (ed.) Tests and Proofs, Lecture Notes in Computer Science, vol. 5668, pp. 90–104. Springer Berlin Heidelberg (2009)
6. Grönniger, H., Rumpe, B.: Considerations and Rationale for a UML System Model. UML 2 Semantics and Applications p. 43 (2009)
7. Jouault, F., Bézivin, J., Kurtev, I.: TCS: a DSL for the specification of textual concrete syntaxes in model engineering. In: Proceedings of the 5th international conference on Generative programming and component engineering. pp. 249–254. ACM (2006)
8. Jouault, F., Teodorov, C., Delatour, J., Le Roux, L., Dhaussy, P.: Transformation de modèles UML vers Fiacre, via les langages intermédiaires tUML et ABCD. Génie logiciel 109, 21–27 (Jun 2014)
9. Liu, S., Liu, Y., André, É., Choppy, C., Sun, J., Wadhwa, B., Dong, J.: A Formal Semantics for Complete UML State Machines with Communications. In: Johnsen, E., Petre, L. (eds.) Integrated Formal Methods, Lecture Notes in Computer Science, vol. 7940, pp. 331–346. Springer Berlin Heidelberg (2013)
10. Lund, M., Refsdal, A., Stølen, K.: 4 Semantics of UML Models for Dynamic Behavior. In: Giese, H., Karsai, G., Lee, E., Rumpe, B., Schätz, B. (eds.) Model-Based Engineering of Embedded Real-Time Systems, Lecture Notes in Computer Science, vol. 6100, pp. 77–103. Springer Berlin Heidelberg (2010)
11. Rumpe, B.: Agile Modellierung mit UML : Codegenerierung, Testfälle, Refactoring. Springer Berlin, 2nd edn. (Jun 2012)
12. Selic, B.: The pragmatics of model-driven development. IEEE Softw. 20(5), 19–25 (Sep 2003)

---

[8] A technique coming from programming environments (see `http://wiki.eclipse.org/FAQ_What_is_a_Quick_Fix%3F`).

Panel Discussion: Proposals for Improving OCL

Achim D. Brucker[1], Tony Clark[2], Carolina Dania[3], Geri Georg[4],
Martin Gogolla[5], Frédéric Jouault[6], Ernest Teniente[7], and Burkhart Wolff[8]

[1] SAP SE, Vincenz-Priessnitz-Str. 1, 76131 Karlsruhe, Germany
achim.brucker@sap.com
[2] Department of Computer Science, Middlesex University, London, UK
t.n.clark@mdx.ac.uk
[3] IMDEA Software Institute, Madrid, Spain
carolina.dania@imdea.org
[4] Computer Science Department, Colorado State University, Fort Collins, USA
georg@cs.colostate.edu
[5] Database Systems Group, University of Bremen, Germany
gogolla@informatik.uni-bremen.de
[6] LUNAM, L'Université Nantes Angers Le Mans
TRAME team, ESEO, Angers, France
frederic.jouault@eseo.fr
[7] Department of Service and Information System Engineering
Universitat Politècnica de Catalunya – BarcelonaTech
teniente@essi.upc.edu
[8] Univ. Paris-Sud, Laboratoire LRI, UMR8623, 91405 Orsay, France
CNRS, 91405 Orsay, France
burkhart.wolff@lri.fr

Abstract. During the panel session at the OCL workshop, the OCL community discussed, stimulated by short presentations by OCL experts, potential future extensions and improvements of the OCL. As such, this panel discussion continued the discussion that started at the OCL meeting in Aachen in 2013 and on which we reported in the proceedings of the last year's OCL workshop.

This collaborative paper, to which each OCL expert contributed one section, summarises the panel discussion as well as describes the suggestions for further improvements in more detail.

1 Introduction

While OCL is nearly 20 years old [6], it is still an evolving language and there is an ongoing effort in academia to improve it. This is also witnessed by the constant updates to the official OMG standards and the current standardisation efforts that will eventually results in OCL version 2.5. Already as a follow up of the last OCL workshop, a number of OCL experts met in November 2013 in Aachen to discuss possible improvements of the OCL (see the report on the OCL meeting in Aachen in the proceedings of the OCL workshop 2013 [2]).

The panel session provided a platform for the OCL community to discuss the presented proposal for improving the OCL as well as to discuss the general future of textual modelling. The following sections, each of them contributed by one expert of the field, discuss the different areas for improvements that were discussed during the panel session.

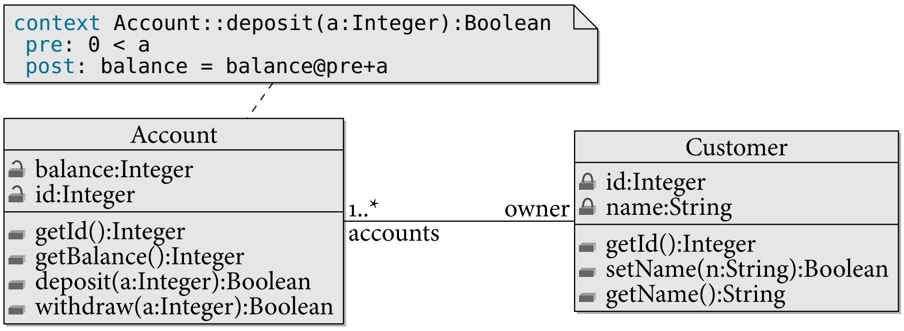
2 Frame Conditions for OCL

Achim D. Brucker. Traditionally, OCL operation contracts do only specify the intended changes to the system state. In general, there is no guarantee that other parts of the system remain unchanged. In particular, the default post condition `true` allows arbitrary changes to the system state.

We suggest to introduce a new method, called `_->modifiesOnly()`, that allows to explicit specify frame conditions, i. e., what can be modified by an OCL operation.

2.1 Motivating Example

When using contracts, or pairs of preconditions and postconditions for state transition there arises the need to specify *exactly* which parts of the system are allowed to be modified and which have to stay unchanged, i. e., we have to specify the frame property of the system. Otherwise, arbitrary relations from pre-states to post-states are allowed. For most applications this is too general: there must be a way to express that parts of the state *do not change* during a system transition, i. e., to specify the frame properties of system transition. As an example, consider Figure 1 with an particular focus on the specification of the operation `deposit` of the class `Account`. This specification only describes which part of the system should change, i. e., the balance of the context object (which is an `Account` object) should be increased. But this is not specified, which parts of the system should remain unchanged, e. g., the `id` of the context object.



```
context Account::deposit(a:Integer):Boolean
 pre: 0 < a
 post: balance = balance@pre+a
```

Fig. 1. Consider a state transition constrained by the operation specification for the operation `deposit`. Obviously, only the attribute `balance` of one specific object should be changed, but how can this be specified?

One solution to solve this frame problem would be an implicit invariability assumption on the meta-level which would somehow express "all things that are not changed explicitly remain unchanged." But this is neither formal nor precise and thus not usable within a formal framework for object-oriented specifications.

Another possibility is to enumerate, in the postcondition of the operation, all path expressions that should remain unchanged, e. g., in our example a first attempt to do so would be:

```
context Account::deposit(a:Integer):Boolean
  post: balance = balance@pre+a
  post: id = id@pre
  post: owner = owner@pre
  post: owner.id = owner@pre.id@pre
  post: owner.name = owner@pre.name@pre
```

But this is also not sufficient, as it would still not describe if objects not related to our context object (of type `Account`) must remain unchanged or not. Enumerating all classes (and attributes) using static path expressions (e. g., `Customer::name = Customer::name@pre`) is tedious and moreover leads to contradictions if the `name` attribute of the owner of the context object should be changed.

Our Proposal. This framing problem is well-known (one of the suggested solutions is,e. g., [5]). We suggest to introduce an OCL method that explicitly allows to specify what might be changed during a system transition. We define

```
(S:Set(OclAny))->modifiesOnly():Boolean
```

where `S` is a set of objects (i. e., a set of `OclAny` objects). This also allows recursive operations collect the set of objects that are potentially changed by a recursive function. Obviously, similar to `@pre` the use of `->modifiesOnly()` is restricted to postconditions.

In our formalisation, called Featherweight OCL [3], we encode the set `S` as a set of object ids (oid). The semantics of the `_->modifiesOnly()` operator is defined such that for any object whose oid is *not* represented in `S` and that is defined in pre and post state, the corresponding object representation will not change in the state transition. A simplified presentation is as follows:

$$I[\![X\texttt{->modifiesOnly()}]\!](\sigma,\sigma') \equiv \begin{cases} \bot & \text{if } X' = \bot \lor \text{null} \in X' \\ \llcorner \forall i \in M.\ \sigma\ i = \sigma'\ i \lrcorner & \text{otherwise}. \end{cases}$$

where $X' = I[\![X]\!](\sigma,\sigma')$ and $M = (\text{dom}\ \sigma \cap \text{dom}\ \sigma') - \{\text{OidOf}\ x |\ x \in \ulcorner X' \urcorner\}$. Thus, if we require in a postcondition `Set{}->modifiesOnly()` and exclude via `_.oclIsNew()` and `_.oclIsDeleted()` the existence of new or deleted objects, the operation is a query in the sense of the OCL standard, i. e., the `isQuery` property is true. So, whenever we have $\tau \vDash X\texttt{->excluding}(s.a)\texttt{->modifiesOnly()}$ and $\tau \vDash X\texttt{->forAll}(x|\texttt{not}(x \doteq s.a))$, we can infer that $\tau \vDash s.a \triangleq s.a\ \texttt{@pre}$.

3 Extending OCL with Functions

Tony Clark. The current OCL standard does not support *functional abstraction.* This is surprising given the origins of OCL and its relationship, with respect to extensive support for collection processing, to functional programming languages such as ML and Haskell. OCL can currently be used to specify operations on classes. It would be useful to extend this notion so that OCL could be used to define a self supporting, albeit high-level and side-effect free, executable system that could be used for a variety of purposes including scripting, simulation, model management, *etc.* Adding functions to OCL is a step in this direction.

In this regard, a *functional abstraction* in OCL will provide a new type of expression that defines an anonymous function comprised of a sequence of named arguments and a body, which is any OCL expression. The denotation of such an expression is a *function* that can be applied to the requisite number of argument values causing the function definition body to be evaluated. Since a function is a value it can be named in the usual way, for example by passing it as an argument to another function or (equivalently) binding in a **let**-expression. Free variables within the body of a function definition will exhibit *lexical scoping*, meaning that the life-time of the function, and any associated free variable values, may outlive that of the binding scope in which it is defined. Since the name of a function is not an intrinsic part of its definition, recursive functions are to be established using a new binding mechanism provided by `let` that is designated as *recursive.*

Having outlined above the characteristics of functions, their use within OCL is motivated as follows:

abstraction Currently OCL lacks a mechanism for abstracting patterns of definitions and then reusing them throughout a system specification. This may take the form of a collection of domain specific functions that provide, for example, arithmetic calculations. Furthermore, the higher-order aspect of functions will facilitate patterns over functions, for example by defining calculations involving sorts where the sort-relationship (alpha-sort, numeric-sort, ascending, descending, *etc.*) is passed as an argument.

modularity Current OCL specifications can be long-winded where expressions contain a great deal of detail. Functions, especially locally defined functions, can help to reduce the complexity both in terms of size and readability. Functions allow parts of a specification can establish a collection of private reusable abstractions. Functions can be the basis of defining both general-purpose and domain-specific library modules for OCL.

iteration OCL provides a string support for processing collections. The iteration processing expressions are built-in to the OCL language when this is not necessary. They can all be defined in terms of a small number of primitive collection operations and recursive functions (as demonstrated by functional languages whose libraries contain a much larger range of collection operators). Languages should strive for both semantic universality and semantic parsimony with regard to their intended domain; currently OCL provides neither.

3.1 Proposal

This section proposes the addition of anonymous function definitions and functions, and associated language support, to OCL. This section provides a brief overview of how this might be achieved.

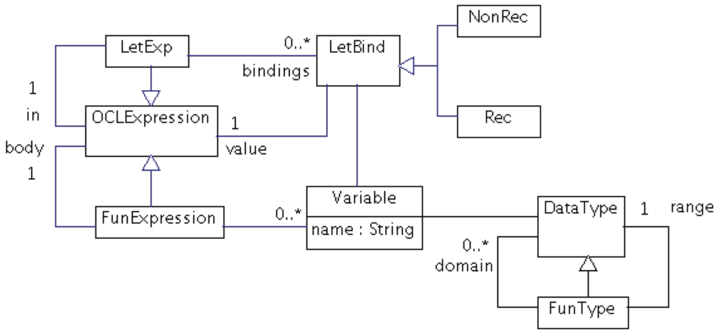


Fig. 2. Abstract Syntax Extension

Types Figure 2 shows the proposed extensions to the OCL abstract syntax model that are necessary to support function definitions. A new type `FunExpression` is introduced as a sub-class of `OCLExpression`, it has any number of (ordered) arguments defined as variables, and a body. The existing `LetExp` class is extended to allow two different types of binding: recursive and non-recursive. The `DataType` class is extended to produce a new data type called `FunType` whose domain and range types describe the argument and body types of a function respectively.

Concrete Syntax OCL functions are supported by a small concrete syntax extension. A function expression including domain and range types can be written as follows (assuming the availability of a function `sqrt`):

```
fun(x:Integer,y:Integer):Integer
  sqrt(x*x + y*y)
end
```

We can name the function:

```
let
  distance = fun(x:Integer,y:Integer):Integer sqrt(x*x + y*y)
  end
in distance(100,200)
end
```

which suggests the following sugar:

```
let distance(x:Integer,y:Integer):Integer = sqrt(x*x + y*y)
in distance(100,200)
end
```

At the top-level of a specification we might allow `distance` to be available everywhere:

```
let distance(x:Integer,y:Integer):Integer = sqrt(x*x + y*y);
```

In the above definition, the name `distance` will not refer to the function being defined (if anything it will refer to a definition in a surrounding scope). To achieve a recursive function, an extra keyword is used:

```
let rec
  size(s:Sequence(T)):Integer = if s->isEmpty
                                then 0
                                else 1 + size(s->rest());
```

3.2 Examples

To see how higher-order features of functional-OCL can be used to good effect, consider the case of OCL without built-in iteration. The `select` expression can be achieved using a function called `select` that is defined in the context of a polymorphic type `Sequence(T)`. The function `select` takes a function `q` as an argument; `q` acts as a predicate on each element of the collection. The function `select` recursively processes the collection and returns a collection containing only those elements that satisfy `q`:

```
context Sequence(T)::select(q:(T)->Boolean):Sequence(T) =
  let s:Sequence(T) = self->rest()->select(q)
      x:T = self->first()
  in if q(x)
     then s->prepend(x)
     else s
     end
  end
```

Now any occurrence of `S->select(e | p)` can be translated to: `S.select(fun(e) p end)` and all other OCL iteration constructs can be treated the same way. This significantly reduces the number of semantic primitives for OCL and provides a basis for new collection processing operations based on higher-order functions, for example:

```
context Sequence(T)::foldr(g:(T,T')->T',x:T'):T' =
  if self->isEmpty
  then x
  else g(self->first(),self->rest().foldr(g,x))
  end
```

Once defined, this can be used as the basis of many different sequence operations:

```
context Sequence(Boolean)::allTrue():Boolean =
  self.foldr(and,true)
context Sequence(Boolean)::anyTrue():Boolean =
  self.foldr(or,false)
context Sequence(Integer)::sum():Integer     =
  self.foldr(+,0)
context Sequence(Integer)::product():Integer =
  self.foldr(*,1)
context Sequence(T)::size():Integer          =
  self.foldr(fun(x) x + 1 end,0)

context Sequence(Sequence(T))::concat():Sequence(T) =
  self.foldr(fun(l1,l2) l1->append(l2) end,Seq{})
context Sequence(T)::reverse:Sequence(T) =
  self.foldr(fun(x,l) l->append(Seq{x}) end,Seq{})
```

4 Implicit Strict Downcasts in OCL Collection Operations

Martin Gogolla. Current OCL allows to select elements of a particular type from a heterogeneous collection and to apply subtype specific operations to the selected elements.

```
Set{4,'VII','IV',7}->
  selectByKind(Integer)->
    collect(i | i*i)
==> Bag{16,49} : Bag(Integer)
```

As an example, consider the above evaluation on a heterogeneous collection with `Integer` and `String` elements. Due to the relatively new operation `selectByKind` this task can be formulated in a more condensed way than in older OCL versions.

```
Set{4,'VII','IV',7}->
  select(x:OclAny | x.oclIsTypeOf(Integer))->
    collect(x | let i:Integer=x.oclAsType(Integer) in i*i)
==> Bag{16,49} : Bag(Integer)
```

As shown above, evaluations of this kind were possible in OCL from the very beginning by employing `select`, type assertions, `collect`, and type downcasts, however more notational overhead was needed when compared to the formulation with `selectByKind`.

The proposal that we put forward here is to reduce the notational overhead even more by allowing explicit downcasts from more general types to more special types in collection operations by explicitly giving a subtype to a variable that is used in the collection operation.

```
Set{4,'VII','IV',7}->
  collect(i:Integer | i*i)
==> Bag{16,49} : Bag(Integer)
```

Starting from types `S` and `G` with `S<G` and a term `COL` evaluating to a collection of type `Collection(G)`, the general translation schema for such explicit downcasts in collection operations would look as indicated below: the central idea is that a call for `colOp` is replaced by a `select` call and a `colOp` call; the variable `s` is typed through the more special type `S` and the OCL expression `expr[s]` uses `s` in contexts where the more special type `S` and not the more general type `G` is expected; the operation `colOp` can be any collection operation, not only as in the above example the collection operation `collect`.

```
COL->colOp(s:S | expr[s])
==>
COL->
  select(x | x.oclIsTypeOf(S))->
    colOp(g:G | let s:S=g.oclAsType(S) in expr[s])
```

In particular applications of this construct in the context of type generalization seem to be useful. For example, assume we have `Female<Person, Male<Person`, the following evaluation would be possible.

```
Set{ada,bob,cyd,dan,eve}->
  collect(f:Female | f.husband.firstName)
==> Bag{'Dan','Dan'}
```

The discussion at the workshop brought up reservations about the use of the colon : at the crucial point of giving a subtype to the variable. In order to avoid accidental use of the subtyping mechanism, using a new, differentiating syntax was discussed. Some proposals are indicated below. The lasts syntax proposal employing brackets are referring to the syntax proposed for patterns matching (inspired from Haskell).

```
Set{ada,bob,cyd,dan,eve}->
  collect(f<:Female | f.husband.firstName)

Set{ada,bob,cyd,dan,eve}->
  collect(f@Female{} | f.husband.firstName)

Set{ada,bob,cyd,dan,eve}->
  collect(f:Female{} | f.husband.firstName)
```

5 Active Operations for OCL

Frédéric Jouault. Before its 2.0 version (when it was still defined as a part of UML [8]), OCL only had three kinds of constraints: `inv` for classifier invariants, as well as `pre` and `post` respectively for preconditions and postconditions of operations[9]. Starting with version 2.0, additional kinds of constraints appeared: `body` to implement (side-effect free) operations, as well as `init` and `derive` respectively to specify initial values, and to implement derived features.

A derived feature is a feature (attribute or association end), which has a value that is computed from the values of other features. The fact that a feature is derived is specified in a class diagram (e.g., in UML), not in OCL. In general, the value of a derived feature is specified in OCL as an invariant. However, the computation of the value of a derived feature is not always trivial from an invariant. Consequently, `derive` constraints were introduced in the OCL specification as a special case of invariants. They work by specifying an expression that evaluates to the value of the derived feature. The computation of the value of a derived feature becomes as simple as evaluating that expression.

Although the more recent `derive` kind of constraint is useful, it notably does not address the 3 following issues that arise when working with derived features:

– **Changeability** is limited to read-only access. Writable derived features must typically be achieved by actual implementation (e.g., in Java), not by modeling in OCL.
– **Observability** is generally not possible because derived features are computed by evaluating the specified OCL expressions, which typically happens on-demand. Arguably, this is more an implementation issue than a modeling one. However, it is a real problem for modeling tools. For instance, a model editor that displays the value of a derived feature cannot directly listen for its changes.
– **Direct use of invariants** to specify derived features should also be possible, but is generally not supported by tools. When `derive` is used instead of `inv`, the person writing the constraint must decide how to compute the derived features. Ideally, tools should be able to do this from more general invariants.

In this section, we propose to extend OCL with active operations [1]. Basically, active operations enable incremental synchronization of collections. In some cases, bidirectionality is even possible.

Here are three concrete benefits of integrating active operations in OCL:

– **Changeability** is achieved by relying on bidirectionality of active operations. This works independently of whether `inv` or `derive` is used.
– **Observability** is possible because active operations incrementally update derived features as soon as there is a change in the values they depend on. This is also independant of the use of `inv` or `derive`.
– **Direct use of invariants** becomes possible without having to rewrite them into `derive` constraints (whether manually or automatically). This means that `inv` may be used instead of `derive`.

Active operations perform their work by producing side-effects on models: some collections get updated when other collections are changed. We believe they

---

[9] `def` "definition" constraints enable expression reuse but do not constrain models.

are nonetheless compatible with OCL because: only "when OCL expressions are evaluated, they do not have side effects" [9], but active operations do not work by evaluating OCL expressions. With active operations, OCL constraints are used in way that is similar to how constraints are used in constraints programming: they specify the form of desired solutions, not how to compute them.

5.1 Motivating Example

Consider Figure 3: a *Transporter* handles several *Transport*s, each associated to a *Truck* and a *Driver*, which are two kinds of *Resource*s. The set of all resources used by a *Transporter* is captured as the *resources* association. Finally, a *Transporter* has three derived features: *derivedResources*, *trucks*, and *drivers*. Note that from a modeling point of view *derivedResources* is redundant with *resources*. However, from a pedagogical point of view we need both derived and non-derived versions of the same relation.

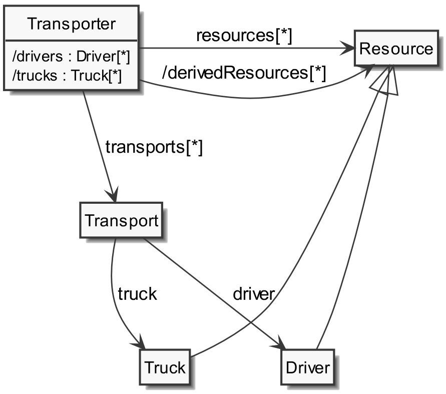


Fig. 3. Transporter class diagram

Listing 1.1 gives a set of OCL constraints over the class diagram depicted in Figure 3. Constraint C1 specifies with an invariant that all *derivedResources* of a transporter must be used in at least one transport. Constraint C2 specifies how the *derivedResources* derived feature can be computed. One can observe that C1 logically implies C2. Nonetheless, C2 must generally be specified so that tools may actually be able to compute the value of the derived feature. With active operations, the more general C1 is enough and C2 is redundant. Nonetheless, C2 would also work without C1. Note that in this case *derivedResources* cannot trivially be made writable because adding a resource to a transporter may then require the creation of a new instance of class *Transport*. This may be captured in the class diagram by specifying *derivedResources* as read-only. It should be noted that UML also provides a specific built-in mechanism for derived unions, which is roughly equivalent to what we are doing here with C1. However, UML specifies that derived unions must be read-only whereas we can do better with active operations.

Similarly, constraint C3 specifies with an invariant that non-derived *resources* correspond to the union of derived *drivers* and *trucks*. Constraint C4 and C5 respectively specify expressions that can be used to directly compute the values of *drivers* and *trucks*. One can observe that C3 logically implies C4, and C5. Nonetheless, C4 and C5 must generally be specified so that tools may actually be able to compute the values of the derived features. With active operations, the more general C3 is enough and C4 and C5 are redundant (but would still work). Additionally, C3 is usable in both directions: *resources*, *drivers*, and *trucks* are all writable features. Moreover, even if C3 did not exist, C4 and C5 would be usable bidirectionaly to update *resources* when either *drivers* or *trucks* was updated.

```
-- [C1] all derivedResources must be used by transports
context Transporter inv:
self.derivedResources = self.transports.driver->union(
        self.transports.truck)

-- [C2] implementation of /derivedResources derived feature
context Transporter::derivedResources : OrderedSet(Resource)
derive: self.transports.driver->union(self.transports.truck)

-- [C3] resources are the union of drivers and trucks
context Transporter inv:
self.resources = self.drivers->union(self.trucks)

-- [C4] implementation of /drivers derived feature
context Transporter::drivers : OrderedSet(Driver)
derive: self.resources->select(e | e.oclIsKindOf(Truck))

-- [C5] implementation of /trucks derived feature
context Transporter::trucks : OrderedSet(Truck)
derive: self.resources->select(e | e.oclIsKindOf(Truck))
```

Listing 1.1. OCL constraints for the class diagram of Figure 3

We have seen above that with active operations, not only are C1 and C3 enough, but they are also enough for the tools to be actually able to compute the values of derived features. Concretely, here is how active operations work for C3:

- **Initialization** consists in setting *drivers* and *trucks* to appropriate values by doing something similar to what C4 and C5 specify, but by only relying on information given in C3. Additionally, the active operations engine starts to listen for changes in either *resources*, *drivers*, or *trucks*.
- **Synchronization** is triggered whenever a change is performed, and the model is considered to be in a stable state only after it is done. If *resources*

is modified, then either *drivers* or *trucks* is updated with the new element. If either *drivers* or *trucks* is modified, then *resources* is updated with the new element. Should these features be ordered, active operations may preserve ordering (e.g., by considering that union is equivalent to concatenation).

Once synchronization has ended, the model is again in a stable state, and the invariant satisfied.

5.2 Conclusion

Active operations address concrete shortcomings of OCL when used with derived features. They are compatible with its semantics, even though they require a different execution engine. Algorithms proposed in [1] can be used to implement active operations, and cover most OCL operations on collections. We do not foresee any issue with missing ones.

6 Purpose-specific Fragments of OCL

Ernest Teniente.[10] An important direction on the improvement of OCL should be devoted to analyze how can we make this language broadly used in industry. In this proposal, we suggest to identify purpose-specific fragments of OCL, each of them devoted to a different goal in software development, as a significant step in this direction. Having purpose-specific fragments of OCL would help its learning and understanding while showing up the benefits of its use.

6.1 Motivation

OCL is aimed at defining all relevant aspects of a specification that cannot be stated diagrammatically. It is a formal language, intended to be easy to read and write and it can be used for a number of different purposes: as a query language, to specify invariants in UML class diagrams, to describe pre- and post conditions on operations and methods, to describe guards, to specify target sets for messages and actions, etc. [7].

OCL is proposed to be used in software development, mainly at the initial stages of this process, because it is intended to fill the gap between natural and classical formal languages being understandable but formal at the same way; and it is expected to be widely used in industry because of the advantages it provides to the automation of code generation or to automated reasoning. This is particularly important in the context of model-driven development.

Providing an answer to the question *"How can OCL be improved?"* should take all these issues into account. So, the aim of this contribution is not to propose some additional, missing, feature of the language to make it better in

---

[10] This work has been partially supported by the Ministerio de Ciencia e Innovación under project TIN2011-24747

some sense but trying to contribute to the debate of why OCL is not used (almost) in industry and to propose a possible solution to it.

There are two possible reasons for that: either the industry does not build (UML) models or OCL is too complex already to be understood by people from industry. In the first case, there would still be a long way to go as far as modeling in industry is concerned and making the language more expressive would not solve the problem. In the second, it seems clear that considering additional features of the language would not help people to better understand OCL.

Some issues allow to illustrate the difficulties to understand OCL. For instance, when learning about the use of OCL expressions in UML models in the OCL specification document [7], we find the following sentence: "everywhere in the UML specification where the term expression is used, an OCL expression can be used (...), but other placements are possible too. The meaning of the value, which results from the evaluation of the OCL expression, depends on its placement within the UML model". Does it mean that we need to read the whole UML specification also to know when an OCL expression can be used? How do we know the different values that an expression can take depending on its placement? How can we easily learn that?

Additionally, some aspects are difficult to explain or may even look like contradictory. A couple of examples follow. Being OCL a declarative language, is the *iterate* construct declarative? How can we justify that? When should *tuple types* be used? What are they useful for? Which is the relationship between the expressive power of OCL and that of well-known languages such as relational algebra or first-order logic?

Under these considerations, and bearing in mind that our purpose is getting OCL to be used in industry, my guess is that we should not actually extend OCL further but trying to simplify or to structure it in such a way that it can be better understood. This is further discussed in the next section.

6.2 Purpose-specific Fragments of OCL

One way to improve the understanding of OCL, thus facilitating its adoption by industry, would be to identify *purpose-specific* fragments of the language. Each fragment should be devoted to a particular purpose or use of the language in software development. So, we could identify a fragment to define invariants in UML class diagrams, another one to specify pre- and postconditions of operation contracts, a third one to state model transformations, etc.

Different fragments would share several OCL operators but probably none of them would require the whole set of actual operators because of its particular purpose. Note that we are not advocating for simplyfing the language nor for removing some operators. We are just proposing to structure the language in such a way that understanding each fragment is easier than understanding the whole OCL. Moreover, knowing that the fragment is aimed at an specific purpose would also facilitate knowing the formal semantics of the language and the comparison with other languages for the same purpose.

The definition of the expressions and the operators of each fragment should be performed in an incremental way, going from the basic concepts and the most common patterns found in its purpose to the most complex and specific issues. Thus, for instance, in the fragment devoted to specify invariants in UML class diagrams one could start by showing how to build expressions that specify invariants tied to a contextual instance and obtained through the navigation of binary associations and, afterwards, proceed with more difficult cases like explaining the particular usage of the *allInstances* operator (i.e. when is it strictly required and why) or the navigation through n-ary associations.

Having a purpose-oriented structure of the language would allow also identifying the tools that allow the automation of software development with that purpose in mind. This is particularly important because without the existence of such tools it will be very difficult for industry to adopt the OCL language. In fact, the best way to convince industry about the usefulness of the use of OCL would be to show the economical benefits they would get and also the improvement on the quality of the final product obtained, and this can only be achieved by means of practical tools.

Finally, and is it clearly happens in the Java community, we would also need books so that people can do self-learning of each specific fragment of OCL. Right now, there are only a very few books, rather introductory, explaining how the elements of the OCL language can be used to complement UML models [4, 10, 11]. Moreover, they are ten years old already and aimed at covering the most common usages of OCL thus being too general as far as self-learning for an specific purpose is concerned.

6.3 Conclusion

Simple is better, definitely. So, if we want the OCL language to be widely used in industry we need to structure it in such a way that it is easily understandable, easy to learn and so that the benefits it provides to software development are out of discussion. One way to achieve this is by considering purpose-specific fragments of the language as advocated in this paper. Tools and books for each of the fragments are also required.

7 Patterns in OCL

Burkhart Wolff. Pattern-Matching is a widely used and well-known concept in functional programming leading to concise and readable code. They are particularly valuable for defining model-transformers and compilers, a domain where OCL is prominently used.

With the advent of Tuples (called *records* in the functional programming literature) we could also introduce pattern-matching wherever variables were bound, so in definitions of recursive functions, quantifiers, select-operators, ...

Moreover, we suggest the concept of *shadow classes* which can be associate to each class-definition `A` (allowing objects in a state) a shadow - tuple `A{a, ... , z}` (so: a value) that is amenable to pattern matching.

7.1 Pattern Matching in Collection Types

OCL possesses hidden second-order combinators, implicitly accepting a lambda buried under first-order notation. These are:

```
->iterate          ->exists        ->forall          ->select
->collect          ->any           ->isUnique.
```

For all these constructs, we propose to allow:

```
  S->select( PATTERN  | P  (x1,...,xn))
```

or in the general case:

```
  S->select( PATTERN1  | P1  (x1,...,xn)
           @ ...
           @ PATTERNm  | Pm  (x1,...,xmn))
```

For example:

```
  S->select(Seq{_, 3, a, b, ...} | a >= 15 and a = b)
```

which filters from a Collection of Sequences integers those who have a 3 as second argument, and where the third argument is larger 15 and identical with the forth argument. It can be seen as shortcut for:

```
  S->select(X | X->nth(2) = 3 and X->nth(3) >= 15
                      and X->nth(3) = X->nth(4))
```

Similarly, a construction like:

```
  S->select(Set{3, a, b, ...} | a >= 15 and a = b)
```

could be seen as shortcut for:

```
  S->select(X | X->includes(3) and
                X->exists(a b | a >= 15 and a = b))
```

With respect to Tuples (called usually *records* in functional programming languages), the following notations are possible:

```
  S->select(Tuple{name='mueller',sex=male,age=x, ...} |
                                                 x >= 21)
```

Note that to be on the safe side, we propose to allow the ... notation for unused labels in a tuple, but allow the pattern-match notation only when the tuple type can be completely inferred.

7.2 Shadow Tuples of Classes

The power of the pattern-matching mechanism is further increased if a seamless transition between objects and corresponding tuples is supported. For example:

```
class Employee is Person
+ salary : Integer[0..1]
+ dept_id : Integer [1]
end
```

induces the implicit declaration of the *shadow-tuple*:

```
Employee{salary : Integer;  dept_id : Integer;
         sex: Sex; name: String }
```

(where `Person` provides the remaining attributes `sex` and `name`) which motivates the pattern matching notation:

```
Employee.allInstancesOf()
        ->select(Employee{salary=x,dept_id=5,... } |
                        x <> null and x>2000 )
```

Access to the implicit object id is forbidden; and we suggest to construct shadow-tuples completely, i.e. not producing *partial* tuples or the like which tends to complicate the type inference. The "..." notation is thus only used in patterns, not in declaration of tuples (introducing a concept like extensible records as in Isabelle).

8 Conclusion

The lively discussion both during the panel discussion as well as for each paper that was presented showed again that the OCL community is a very active community. Moreover, it showed that OCL, even though it is a mature language that is widely used, has still areas in which the language can be improved. We all will look forward to upcoming version of the OCL standard.

Acknowledgments. We would like to thank all participants of this years OCL workshop for their active contributions to the discussions at the workshop. This lively discussions are a significant contribution to the success of the OCL workshop series.

References

[1] Beaudoux, O., Blouin, A., Barais, O., Jézéquel, J.M.: Active Operations on Collections. In: Petriu, D., Rouquette, N., Haugen, Ø. (eds.) Model Driven Engineering Languages and Systems, Lecture Notes in Computer Science, vol. 6394, pp. 91–105. Springer Berlin Heidelberg (2010)

[2] Brucker, A.D., Chiorean, D., Clark, T., Demuth, B., Gogolla, M., Plotnikov, D., Rumpe, B., Willink, E.D., Wolff, B.: Report on the Aachen OCL meeting. In: Cabot, J., Gogolla, M., Rath, I., Willink, E. (eds.) Proceedings of the MoDELs 2013 OCL Workshop (OCL 2013). CEUR Workshop Proceedings, vol. 1092, pp. 103–111. ceur-ws.org (2013)

[3] Brucker, A.D., Tuong, F., Wolff, B.: Featherweight ocl: A proposal for a machine-checked formal semantics for ocl 2.5. Archive of Formal Proofs (Jan 2014), `http://afp.sf.net/entries/Featherweight_OCL.shtml`, Formal proof development

[4] Clark, T., Warmer, J. (eds.): Object Modeling with the OCL, The Rationale behind the Object Constraint Language, Lecture Notes in Computer Science, vol. 2263. Springer (2002)

[5] Kosiuczenko, P.: Specification of invariability in ocl. In: Nierstrasz, O., Whittle, J., Harel, D., Reggio, G. (eds.) Model Driven Engineering Languages and Systems (MoDELs). Lecture Notes in Computer Science, vol. 4199, pp. 676–691. Springer-Verlag, Heidelberg (2006)

[6] Object constraint language specification (version 1.1) (Sep 1997), available as OMG document ad/97-08-08

[7] Object Management Group (OMG): Object Constraint Language (UML), version 2.3.1 (2012), `http://www.omg.org/spec/OCL/`

[8] (OMG), O.M.G.: Unified Modeling Language (UML), Version 1.5. `http://www.omg.org/spec/UML/1.5/` (Mar 2003)

[9] (OMG), O.M.G.: Object Constraint Language (OCL), Version 2.4. `http://www.omg.org/spec/OCL/2.4/` (Feb 2014)

[10] Warmer, J., Kleppe, A.: The Object Constraint Language: Precise Modeling with UML. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (1999)

[11] Warmer, J., Kleppe, A.: The Object Constraint Language: Getting Your Models Ready for MDA. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2 edn. (2003)