UnifiedOCL: Achieving System-Wide Constraint Representations

David Weber, Jakub Szymanek, and Moira C. Norrie

Department of Computer Science, ETH Zurich, Switzerland
{weber,norrie}@inf.ethz.ch,jakub.szymanek@alumni.ethz.ch
http://www.globis.ethz.ch

Abstract. Constraint definitions tend to be distributed across the components of an information system using a variety of technology-specific representations. We propose an approach where constraints are managed in a single place using *OCL* with extensions for technology-specific concepts. These constraints are then mapped to technology-specific representations which are validated at runtime. Bi-directional translations of constraint definitions allows existing components to be easily integrated into the system. We present an implementation of the approach and report on a user study with developers from industry and research.

Keywords: constraints, OCL, transformations

1 Introduction

Constraints for basic data validation typically are defined in different components of an information system, including client-side form validation, business logic and databases. Further, the set of applicable constraints may depend on a particular configuration or context. For example, different constraints might apply in the mobile and desktop versions of an application due to differences in the functionality offered. Since constraints are derived from software requirements which may evolve over time, especially with agile methods, it can be challenging for developers to keep track of all constraints and maintain consistency throughout an entire system. It would therefore be highly beneficial if constraint definitions could be managed in a single place and automatically mapped to component-specific implementations for runtime validation.

To achieve this, we propose an approach in line with *Model-Driven Architecture (MDA)* [7, 9] where a technology-independent model serves as a base for technology-specific code generation. In our model, constraints are expressed in UnifiedOCL, a domain specific language which extends OCL with capabilities to represent technology-specific constraints. To cater for the integration of different technologies, the grammar of UnifiedOCL is extensible by means of so-called *labels* which are grouped into dictionaries.

Furthermore, we studied various approaches and technologies of bi-directional translations between various models and representations. This resulted in an extensible toolkit capable of efficiently translating constraints to and from *UnifiedOCL*. As a consequence, the system allows for translation from any source representation to any target representation, which we call multi-translations. Our proof-of-concept implementation supports three technology-specific representations: object-oriented language (*Java*), relational database (*SQL*) and business rules (*Drools*), which let us explore and cover a broad range of constraints.

After discussing the background and related work in Sect. 2, we introduce our approach in Sect. 3. The details of the unified constraint representation are presented along with our *DSL UnifiedOCL* and pluggable label dictionaries in Sect. 4. In Sect. 5, we describe our multi-translations and we report on the user study that we carried out in Sect. 6. Concluding remarks are given in Sect. 7.

2 Background

The *OCL* [16] plays an important role alongside *UML* in model-centric methodologies and enables concepts such as design by contract to be supported. However, the fact that it is platform independent results in the limitation that not all types of constraints that can be defined in technology-specific representations can be expressed directly. For example, the primary key constraint in relational databases has to be represented by a combination of *not null* and *unique* constraints. Further, it cannot express common restrictions on values such as email addresses or number precision which can be handled in Java using `@Email` and `@Digits` annotations, respectively. Additionally, *OCL* is not powerful enough to allow for the validation of numbers that require complex algorithms computing checksums such as the *Luhn algorithm* for a credit card number.

Several approaches have been proposed for translating *OCL* constraints to technology-specific languages. Many of these provide translations to *SQL*, e.g. [11, 12]. Others have proposed translations to Java, e.g. [17] and specifically also to the Java Modeling Language (*JML*) [8], e.g. [5]. Aspect-oriented approaches, e.g. [6], translate design constraints defined in *OCL* into aspects that, when applied to a particular implementation, check the constraints at runtime. A comparison of different approaches to constraint validation based on *OCL* is provided in [1]. Their comparison includes direct translation to implementation languages, use of executable assertion languages, and use of aspect-oriented programming languages.

Some projects support translations from *OCL* to any form of assertion, e.g. [2, 10]. Moiseev et al. [10] use the same *MDA* approach as we do, but with the goal of showing how structural similarities can be used to generate additional translations with minimum effort.

Cosentino and Martínez [3] address reverse engineering by translating relational schemas into *OCL* expressions. Since *OCL* constraints are not *context-free*, they also generate the associated *UML* class diagram.

Shimba et al. [13] is one of the few projects that handle bi-directional translations, in their case between *OCL* and *JML*. Like us, their implementation is based on *Eclipse*[1] and *Xtext*[2]. However, although they mention Query / View / Transformation Operational (*QVT*)[3] and the ATL Transformation Language (*ATL*)[4], they do not explicitly state if they use it for writing translation rules. We used *QVT* for only one case, using *Xtend*[5] for all others.

[1] `https://eclipse.org/`
[2] `http://www.eclipse.org/Xtext/`
[3] `https://projects.eclipse.org/projects/modeling.mmt.qvt-oml`
[4] `https://wiki.eclipse.org/ATL`
[5] `https://eclipse.org/xtend/`

To the best of our knowledge, ours is the first approach that supports bi-directional translations between an *OCL-like* platform-independent constraint model and multiple technology-specific representations. This is beneficial if legacy systems want to move from one constraint representation to another or if a software developer prefers not to use a specific representation.

3 Approach

Fig. 1 presents an overview of our approach. The various technology-specific constraint representations are depicted at the bottom (*JavaBeans Validation*, *SQL*, *Drools*). Each representation could be used in one or more components of the information system.

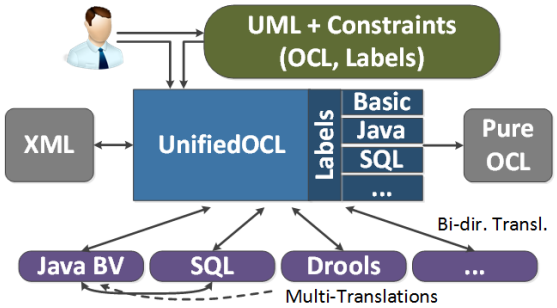

Fig. 1. Approach overview

UnifiedOCL is implemented as a new Domain Specific Language (*DSL*) that allows developers to define a wide variety of constraints using a textual representation that can be extended with labels to provide a compact way of specifying common constraints. This makes it easy for non-programmers to understand.

It combines structural domain information (*UML*), constraint definitions from *OCL* and a set of labels which either represent constraints not available in *OCL* (such as primary key in *SQL*) or simply provide a more convenient representation (such as email and credit card constraints). The syntax and constraint representation capabilities of *UnifiedOCL* will be presented in Sect. 4, while details of the constraint translations will be given in Sect. 5.

In our approach, it is possible to model an information system using a *UML* editor, enhancing the model with constraints defined in the standard *OCL* notation or with our label notation as depicted at the top of Fig. 1. This *UML* model can then be converted to a *UnifiedOCL* representation. Nonetheless, it is also possible to create and modify the *UnifiedOCL* representation in a text editor independent of a *UML* model.

By additionally introducing individual translations from all technology-specific representations to *UnifiedOCL* as indicated by the bi-directional arrows in Fig. 1, it is possible to translate between any two constraint representations in an information system. To add an extension for a technology-specific representation, only one single bi-directional translation to and from *UnifiedOCL* has to be provided to achieve multi-translations as depicted at the bottom of Fig. 1. This is possible since *UnifiedOCL* is powerful enough to express most of the constraints from various representations. Furthermore, it is possible to translate *UnifiedOCL* to pure *OCL* and exchange *UnifiedOCL* in an *XML* format.

4 Unified Constraint Representations

UnifiedOCL specifies both the constraints and the contextual data structure in which constraints exist, such as an *SQL* table schema or a Java class. An example is presented in List. 1 where *OCL* together with our concept of labels are used to define constraints.

List. 1. Example of *UnifiedOCL*

```
package org.example {
 public abstract class Employee {
  public attribute id: Integer {primarykey};
  public attribute name: String {notnull};
  public attribute birthDate: Date {past};
  public attribute email: String {email};
  public attribute salary: Real {range(min=2500, max=5300)};

  public reference subordinates: Employee[0..*]{unique};

  invariant IsJunior: self.age<25;
  invariant PersonId: not self.id.oclIsUndefined()
      and Person.allInstances()
      ->forAll(a : Person|a<>self implies a.id<>self.id);

  public operation checkIn (time: in Date,
      checkedIn: inout Boolean): String {
   precondition NotAlreadyCheckedIn: checkedIn=false;
   body CheckInBody: checkedIn=true;
   postcondition SuccessfullyCheckedIn: result=true;
  };
...
```

In the given example in List. 1 the data structure is `class Employee` in `package` (`org.example`) with the attributes (`id`, `name`, etc.) of data types (`Integer`, `String`, etc.). *UnifiedOCL* labels are used to define constraints associated with these. For example, the attribute `id` is labelled as a `primarykey` based on the *SQL* concept. The labels `past` and `email` define pattern matches not supported in *OCL*, while the `notnull`, `range` and `unique` labels provide a convenient way of defining the appropriate constraints.

Additionally, the `reference` (`subordinates`) has a label `unique` to disallow duplicates along with cardinality constraints specifying that any number of subordinates can exist, where the syntax `[0..*]` is part of the *UnifiedOCL* grammar. *UnifiedOCL* also allows directionalities to be specified for the parameters of operations as shown in the definition of the `checkIn` operation. Pre-, body- and post-conditions are also defined for this operation.

In contrast, the `invariant` constraints `IsJunior` and `PersonId` are defined in *OCL*. We have included a duplicate definition of the `PersonId` invariant using the `primarykey` label just to emphasise the convenience of using *UnifiedOCL* labels over *OCL* notation. With labels, the developer only has to name commonly used constraints and not care about the implementation. The technology-specific

representation and validation of these constraints is the concern of the constraint translation which is discussed in the next section.

Labels also provide a way of extending the language with technology-specific concepts since the set of labels may be expanded. A list of all labels defined within our current implementation together with their parameters can be found in [15, p.163-165].

UnifiedOCL is based on the *OCLinEcore* syntax which allows an *EMF Ecore model* [14] to be combined with *OCL* statements. The *OCLinEcore* grammar extends the *EssentialOCL* [4] grammar, which allows *OCL* expressions to be specified but has no relation to the *Ecore* (or any other) data model. We derived *UnifiedOCL* by extending the *EssentialOCL* language and reusing the *OCLinEcore* language components and *Ecore* metamodel. We changed the metamodel by extending *OCLinEcore* concepts with attributes or relationships required to reflect the structure of a *UML* class diagram. For example, this involved introducing 1) a grammar element to reflect the concept of encapsulation levels, 2) simple exceptions, 3) a directionality meaning for method parameters, 4) user defined primitive types, 5) a `Date` built-in primitive type, and 6) a grammar element to allow operations to be specified as multithreaded. Additionally, and most important, we introduced support for the special constraint representation labels described above.

5 Constraint Translations

We distinguish between a *formal constraint definition* specified in *UnifiedOCL* and a *technology-specific representation* which is dependent on the particular programming language used for validation code. Bi-directional translations are concerned with generating the validation code from the formal constraint definition and vice versa.

For example, assume that a `Food` entity has a `name` attribute of type string which must be at least of length 3. The *formal constraint definition* in *UnifiedOCL* is shown in List. 2.

List. 2. *UnifiedOCL* translation example

```
package org.example {
  public abstract class Food {
    public attribute name: String {length(min=3)};
  }
}
```

The *technology-specific representations* for *SQL*, *Bean Validation* (*BV*) and *Drools* are given in List. 3, 4 and 5, respectively:

List. 3. *SQL* translation example

```
CREATE TABLE Food (
  name VARCHAR NOT NULL,
  CONSTRAINT ck_food_name CHECK (length(name::text) >= 3)
);
```

List. 4. Java *BV* translation example

```
package org.example;
public class Food {
  @Length(min = 3)
  public String name;
}
```

List. 5. *Drools* translation example

```
rule "Food_name_minlength"
when Food (name.length < 3)
then
  ...
end;
```

All translations, except for pure *OCL* extractions, use the model-to-text *M2T* [18] approach and consist of three main steps: 1) obtaining a traversable in-memory model instance from the source file(s), 2) model discovery and analysis and 3) performing a model to text serialisation.

Three different approaches have been used for generating an in-memory model such as an abstract syntax tree (*AST*). In some cases, such as *Drools*, an existing parser that accompanies the technology was used. For *SQL*, a parser was generated with the help of the modeling framework *Xtext* which is based on the *ANTLR*[6] parser. Bean Validation is an example where a model discovery tool such as *MoDisco*[7] was used.

Usually, the entire model needs to be analysed before serialisation can begin. For example, constraints in *SQL* can be defined in various places including as part of a column definition, at the table level or as an `ALTER TABLE` statement defined outside of the table, possibly in a different script. Therefore, an intermediary model is generated before being converted to the target textual representation.

When analysing a model, we identify constraint occurrences with regular expressions and create abstract constraint representations (*ACR*s) containing all information about the constraint in a technology independent way. Dictionaries define the mapping of positions within regular expressions to constraint parameters. An *ACR* defines the location, type, names, parameters and values of a constraint. To generate a target representation, we use specific target representation producers (*TRP*s) which are also defined in dictionaries and able to serialise the *ACR*s to the target textual representation.

In the *JavaBeans Validation* example of List. 4, the technology-specific constraint `@Length(min = 3)` would be matched with an abstract *size* constraint from our *Java2UnifiedOCL* dictionary. The resulting *ACR* would then have the location `org.example.Food.name`, the type *size*, a unique name and the *lowerBound* parameter with a value of 3. Our *General2UnifiedOCL* dictionary can be used to translate any *ACR* to a specific *UnifiedOCL* representation and would be used to generate the representation in List. 2. Similarly, the dictionary could also be used for the opposite translation from *UnifiedOCL* to *JavaBeans Validation*.

The extraction of *OCL* statements from *UnifiedOCL* increases the usefulness of our system since tools such as *DresdenOCL*[8] allow objects such as JavaBeans to be validated based on the pure *OCL* specification.

[6] `http://www.antlr.org/`
[7] `https://eclipse.org/MoDisco/`
[8] `http://www.dresden-ocl.org/index.php/DresdenOCL`

6 Evaluation

We conducted a user study with 20 software engineers from both research and industry (15 male, 5 female). 50% had studied or worked in a university at some time and 75% had worked in industry.

The participants had to solve three tasks with the help of our tool. Before each task, additional features of the tool relevant to the task were explained. A feedback questionnaire based on a 5-point Likert-type scale ('totally disagree' to 'totally agree') was used. In this summary of results, we combine the first two values into 'disagree' and the last two into 'agree'.

In the first task, participants had to convert a *UML* diagram with two simple, connected entities without any constraints and a total of eight attributes into their choice of *Java*, *SQL* or *Drools* code. They were then presented with the same *UML* diagram with seven constraint definitions in *UnifiedOCL* label notation, without an explanation of the notation, and were required to extend the code accordingly. Finally, they had to generate the same code using our tool and edit it if it did not meet their expectations.

For the second task, they were given a short introduction to *UnifiedOCL* and asked to add a field to one of the classes along with a specific constraint. The third task, required them to combine a definition in Java with one in *SQL* to give a single definition in Java, *SQL* or *UnifiedOCL* using our tool.

85% of the participants chose Java in task 1 and 15% *SQL*. Only 30% were able to manually solve the task completely, even though 80% of all users had a good or very good understanding of *UML* and almost half (43%) of those with partial solutions declared their knowledge in the selected technology as very good.

The average time users spent writing initial code was 185 sec (std. dev. 70 sec), while the average for extending it was 224 sec (std. dev. 73 sec). The average time required using our tool was significantly less at 46 sec (std. dev. 21 sec). Moreover, all agreed that the generated code met their expectations, with only five of them editing it slightly to adjust formatting, organise imports, or alter names. All agreed that the tool was easy to learn and use.

Task two was successfully solved by almost all participants (90% knew how to define the field and 85% the constraint) with failures due to a misunderstanding about what to do rather than not knowing how to do it. 90% of users agreed that the *UnifiedOCL* syntax is intuitive. Similarly, almost all (75%) agreed that they would remember the *UnifiedOCL* syntax required to solve this task.

In task three, there were no errors in the use of the system. Most users (60%) chose *UnifiedOCL* as the common format used for the comparison with stated reasons including '*UnifiedOCL* can represent all constraints whereas Java and *SQL* will have some problems with certain constraints' and 'this translation is easier and more obvious'. Of those selecting conversion to Java or *SQL*, most justified it in terms of familiarity, with one stating that *UnifiedOCL* would be the preferred choice if they had more experience with it.

All participants agreed that the tool was efficient and convenient to use, and 85% were satisfied with the results. Those who were not satisfied were missing

some sort of order of the attributes in the target representation. We decided to maintain the same order of attributes as in the source representation, but nevertheless it is one of the extensions that should be considered since it was mentioned by seven participants (35%).

Full details of the results are provided in [15, p.136-143].

7 Conclusions

We have presented an approach that allows constraints validated in different components of an information system to be managed in a central repository, with technology-specific validation code generated automatically. Constraints are specified in *UnifiedOCL* using a mix of *OCL* and *UnifiedOCL* labels which provide a shorthand for common technology-specific constraints. *UnifiedOCL* is extensible as new labels can be introduced at any time.

References

1. C. Avila, A. Sarcar, Y. Cheon, and C. Yeep. Runtime Constraint Checking Approaches for OCL, A Critical Comparison. In *SEKE*, 2010.
2. L. Baresi and M. Young. Toward Translating Design Constraints to Run-Time Assertions. *Electr. Notes Theor. Comput. Sci.*, 116, 2005.
3. V. Cosentino and S. Martínez. Extracting UML/OCL Integrity Constraints and Derived Types from Relational Databases. In *MoDELS Int'l Workshops*, 2013.
4. EssentialOCL. `http://help.eclipse.org/juno/index.jsp?topic=%2Forg.eclipse.ocl.doc%2Fhelp%2FEssentialOCL.html`. Accessed: 2016-04-04.
5. A. Hamie. Using Patterns to Map OCL Constraints to JML Specifications. In *MODELSWARD*, 2014.
6. M.U. Khan, N. Arshad, M.Z. Iqbal, and H. Umar. AspectOCL: Extending OCL for Crosscutting Constraints. In *ECMFA*, 2015.
7. A. Kleppe, J. Warmer, and W. Bast. *MDA explained - the Model Driven Architecture: practice and promise*. Addison-Wesley, 2003.
8. G.T. Leavens et al. JML: A Notation for Detailed Design. In *Behavioral Specifications of Businesses and Systems*, pages 175–188. Springer, 1999.
9. S.J. Mellor, K. Scott, A. Uhl, and D. Weise. Model-driven architecture. In *Advances in Object-Oriented Information Systems*. Springer, 2002.
10. R. Moiseev, S. Hayashi, and M. Saeki. Using Hierarchical Transformation to Generate Assertion Code from OCL Constraints. *IEICE Transactions*, 94-D(3), 2011.
11. N. Obrenovic, A. Popovic, S. Aleksic, and I. Lukovic. Transformations of Check Constraint PIM Specifications. *Computing and Informatics*, 31(5), 2012.
12. X. Oriol and E. Teniente. Incremental Checking of OCL Constraints with Aggregates Through SQL. In *Int'l Conf. on Conceptual Modeling - ER*, 2015.
13. H. Shimba, K. Hanada, K. Okano, and S. Kusumoto. Bidirectional Translation between OCL and JML for Round-Trip Engineering. In *APSEC*, 2013.
14. D. Steinberg, F. Budinsky, M. Paternostro, and E. Merks. *EMF: Eclipse Modeling Framework 2.0*. Addison-Wesley Professional, 2nd edition, 2009.
15. J. Szymanek. Achieving Unified Data Quality Representation by Constraints Transformation. `http://dx.doi.org/10.3929/ethz-a-010510131`, 2015.
16. J. Warmer and A. Kleppe. *The Object Constraint Language: Getting Your Models Ready for MDA*. Addison-Wesley Longman Publishing Co., Inc., 2nd edition, 2003.
17. C. Wilke. Java Code Generation for Dresden OCL2 for Eclipse. Technische Universität Dresden, Germany, 2009.
18. M. Wimmer and L. Burgueño. Testing M2T/T2M Transformations. In *MODELS*, 2013.