IEICE TRANS. INF. & SYST., VOL.E94–D, NO.3 MARCH 2011

PAPER

Using Hierarchical Transformation to Generate Assertion Code from OCL Constraints**

Rodion MOISEEV[†*], **Shinpei HAYASHI**[†a)], ***Nonmembers***, ***and*** **Motoshi SAEKI**[†], ***Member***

SUMMARY Object Constraint Language (OCL) is frequently applied in software development for stipulating formal constraints on software models. Its platform-independent characteristic allows for wide usage during the design phase. However, application in platform-specific processes, such as coding, is less obvious because it requires usage of bespoke tools for that platform. In this paper we propose an approach to generate assertion code for OCL constraints for multiple platform specific languages, using a unified framework based on structural similarities of programming languages. We have succeeded in automating the process of assertion code generation for four different languages using our tool. To show effectiveness of our approach in terms of development effort, an experiment was carried out and summarised.

key words: *OCL, constraints, assertion code, programming languages*

1. Introduction

Model-centric methodologies for software development such as OMG's Model Driven Architecture (MDA) [2] are becoming significant in academia and industry, and Unified Modeling Language (UML) and Object Constraint Language (OCL) [3] play an important role in these methodologies. For instance, UML class diagrams express the structural design of the system, where OCL specifies properties that must be satisfied at a certain time in the system. Figure 1 illustrates how class diagrams and OCL descriptions could be used during a development. The code skeleton is automatically generated from the class diagram, e.g., by using EclipseUML [4], which is then used by developers to complete the implementation. The OCL specifications can then be used to generate code for checking the system at run-time, and/or unit test code. We will generally refer to it as *assertion code* hereafter. If we take Java as a possible implementation language, we should translate the OCL specification into Java-based assertion code. Similarly, for Python and Perl, we need translators from OCL into Python and Perl respectively. This means that we should have an OCL translator for each implementation language. However, a tremendous amount of effort is necessary to develop an OCL translator from scratch for each language.

Consider another more concrete motivating scenario with UML and OCL depicted in Fig. 2 in a distributed system like a client/server system. Assume we have designed an application in UML, defined an interface to it with a method `doQuery` and stipulated several constraints on this method in OCL as shown at the bottom of the figure. The constraints on the method `doQuery` form a contract for accessing that method. The contract is a statement: "Caller must provide the callee an argument `q : Query` that is, at the least not `null`, for the return value to adhere to the range from `0` to the maximum value specified by `MAX_DATA`, in case of successful execution." After implementing our application in Perl, we decide that we want the interface to be accessible by clients over the network without restricting the client implementation language. Therefore clients can be implemented in any arbitrary language, e.g., Python or Java. Now the difficulty is that the contract in our case obliges the caller to check the pre-condition (`q != null`) and obliges the callee to warrant the post-conditions (`result >= 0` etc.), which therefore means that there is a need for an OCL evaluator or checker to be present on all server and clients, which can be implemented in different implementation languages.

The motivation scenario describes a possible usage of OCL that spans implementations in different languages, which imposes a need for OCL translators for multiple languages within one design. In such cases using OCL is desirable since OCL is independent of the implementation technology, such as implementation languages and platforms. Most of currently available UML modelling tools can generate code from UML models, some of which also have the facility to input constraints in OCL. However, not many can make use of those constraints during the implementation stage (the dotted section of Fig. 1). Also, investigating the current approaches (see Sect. 5), we can conclude that currently there exists no approach suitable for working with OCL constraints on the implementation level for multiple languages. Thus we should have a technique to develop an OCL translator for each implementation language with least possible effort.

In this paper, we propose an approach to allow developers to make use of OCL from UML diagrams and to check developers' programmes, for multiple implementation languages. This will be achieved by translating constraints in OCL into their equivalent assertion code in the target language (e.g., Java, C, C++ or Python). Our OCL translator is based on model transformation. One of the advantages of model transformation is the ability to reuse transformation

Manuscript received May 19, 2010.
Manuscript revised October 12, 2010.

[†]The authors are with the Department of Computer Science, Graduate School of Information Science and Engineering, Tokyo Institute of Technology, Tokyo, 152–8552 Japan.
[*]Presently, with Infoscience Corporation.

**This paper is an extended version based on our previous paper published in MoDELS 2009 [1].

a) E-mail: hayashi@se.cs.titech.ac.jp

DOI: 10.1587/transinf.E94.D.612

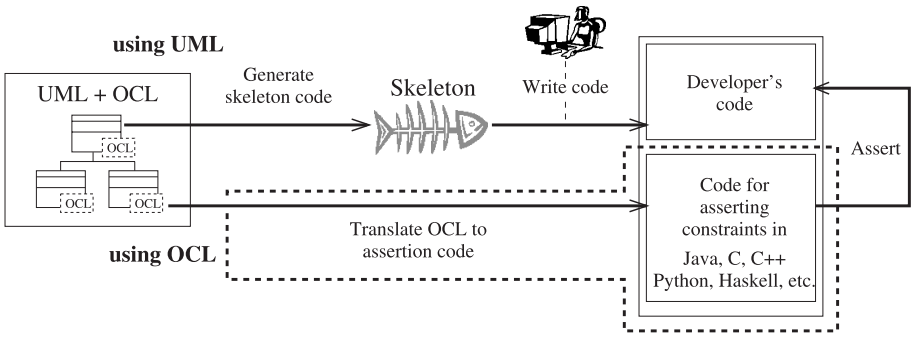


Fig. 1 UML and OCL usage overview.

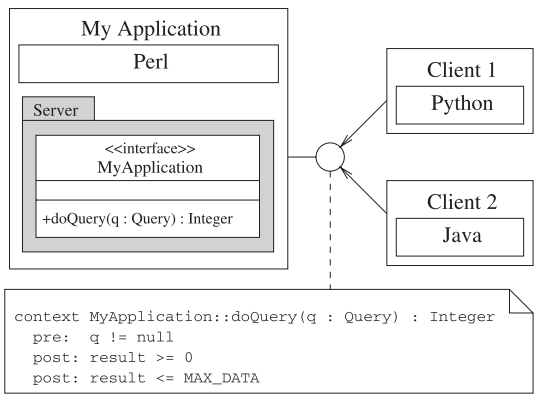


Fig. 2 Motivating scenario: interface with OCL constraints.

rules when developing similar application software, more concretely, transforming semantically similar models. Specifications of the OCL and implementation languages can be modelled as an abstract syntax tree (AST). The transformation rules between these two models can be used to achieve generation of assertion code that can be executed on the implementation language platform. Consider two different implementation languages that could be possible targets for generating OCL assertions. If these two languages are similar, e.g., both of them are imperative programming languages, some of their transformation rules can be shared, so that we can reduce the efforts to design the transformation rules. Therefore we can mitigate the above problem mentioned in the last paragraph, i.e., larger efforts to develop an OCL translator for each implementation language.

The approach focuses on making sure that most of the OCL translation can be done within one framework independent of the target implementation language. On the other hand, because of this lack of dependency, creating an OCL translator for an additional target language requires little understanding of OCL itself on behalf of the developer, which is important for cases when software designer (working with OCL) and programmer (working with implementation languages) are not the same person. The translation process is hierarchically managed which makes it easily modifiable and extendible. It also allows AOP-like [5] modifications at higher (more abstract) levels independently of concrete language implementations, where a good example of useful modifications includes: logging, constraint checking, etc. The approach was designed to make the following improvements upon existing approaches [6]–[8] (compared in Sect. 5):

Extendibility:

Our approach was designed to be extendible to multiple languages, by providing the following advantages:

- Low efforts when creating OCL assertion code generators for languages with no OCL support.
- Minimum efforts when creating support for a new language that is an extension or a modification of an existing language.

Maintainability:

By using a semantic hierarchy, each individual module (step in the hierarchy) is naturally decoupled from others, which improves the maintainability in the following way:

- Modifications tend to be more cohesive with respect to individual modules, hence lowering modification costs.

Understandability:

Our approach attempts to reduce the required knowledge for implementing an OCL evaluator because:

- Developing an OCL evaluator for a concrete language only requires to know that language, and the associated DL, which should be semantically close to the target language. Therefore OCL assertion code generators can be created with minimum understanding of OCL concepts.
- On opposite, OCL related work can be performed at more abstract levels, requiring little knowledge of the concrete implementation languages down the hierarchy.

The main contributions of the paper can be summarised as follows:

- providing a technique to develop an OCL assertion code generator using model transformation with less efforts and
- showing an evaluation result with Rozyn, the transformation tool we have developed.

The evaluation shows that by using our approach we can

develop OCL translators for four different implementation languages with less transformation rules (hence, less effort), opposed to developing each translator separately.

The rest of this paper is organised as follows. In the next section, we describe our approach. Section 3 describes its architecture, pointing out how it realises the contributions we have claimed. In Sect. 4, we cover an evaluation experiment for our approach and summarise the results. Some related work is covered in Sect. 5, followed by conclusion in Sect. 6 and future work described in Sect. 7.

2. Proposed Approach: Basic Idea

Our approach comprises a framework that allows developers to easily and with minimum effort create a generator of OCL assertion code for multiple text-based languages by reusing the mappings to OCL from other languages' structural and semantic concepts.

In order to understand the main concept behind our approach, consider the sample code shown in Fig. 3. We have a sample Python code in Fig. 3 (a) that checks each `apple` in a collection of apples, `basket`, to see whether it is of red colour. The above semantics are expressed in terms of a *for*-loop and an *if*-statement nested inside it. If you further consider a sample Java code in Fig. 3 (b), you will find that even though syntactically it looks somewhat different, semantically and structure-wise it is nearly identical. First, the idea of a *for*-loop for iterating over the collection, and an *if*-statement for doing logical checks is the same as in the Python example. Also the *if*-statement is again nested inside the *for*-loop. These show that Python and Java programming languages resemble in their conceptual vocabulary and in structure, even though the detailed syntax is different.

In fact, we can make similar observations with most imperative programming languages, including Java, Python, Ruby, Perl, C++ and C#. Since all of these languages are based on the same *imperative* programming paradigm, they will contain basic flow control structures such as *for*-loops, *if*-statements, sub-procedures, or other structures expressible in terms of the basic ones. Because of this similarity, we can describe *for*-loops such as the one in Figs. 3 (a) and 3 (b), with a pseudo-language description which captures the semantics of the performed action, shown in Fig. 3 (c). What we are trying to say is that, imperative languages all bear similarities in their semantics originally and therefore share a lot of common programming structures. Of course, other language types, like *functional* languages also all share common structures, since most of them were designed to solve the same problem. Some languages within a certain type, e.g., imperative, may have discrepancies from other languages of the same type, such as having additional constructs or lacking some. In such cases, the developer could extend the main pseudo-language with another layer and provide the required semantic translation between the two pseudo-languages. If such semantic translation is difficult or impossible, extending from a different language type, or creating a new language type would be a better choice.

If we can extract common language features from all languages that fall under a particular category, such as imperative languages, we could create, for example, an imperative pseudo-language that captured all of the common constructs available in imperative programming languages. Such imperative pseudo-language could be used to describe behaviour of OCL constructs in terms of imperative constructs. By doing this, not only we can make the translation to the target imperative language easier, but also alleviate the need to completely comprehend every OCL expression. Based on this idea, in our approach we define a hierarchy, based on structural similarities of commonly available languages. Refer to Fig. 4 for an example of such hierarchy. Some languages can be further subdivided into sub-hierarchies to capture similarities that are more fine-grained, and thus more specific to particular imperative languages. An example of such languages would be Python and Java, which both have *foreach*-loops. Such hierarchy allows us to describe most complex OCL concepts in terms of intermediary pseudo-languages (e.g., imperative or functional pseudo-languages) in one or more steps, therefore abstracting OCL concepts away from implementation languages. This means that at the lower, more concrete levels, the developer will only need to provide details specific to language syntax and grammar to complete the mapping.

```
for apple in basket:
    if apple.colour == "red":
        ...
```

(a) Sample Python code.

```
for (Iterator i = basket.iterator();
        i.hasNext(); ) {
    Apple apple = (Apple) i.next();
    if (apple.getColour().equals("red")) {
        ...
```

(b) Sample Java code.

```
For Each apple In basket Do:
    If apple.colour Is "red" Then
        ...
```

(c) Pseudo-description of the same action.

Fig. 3 Language similarities extraction.

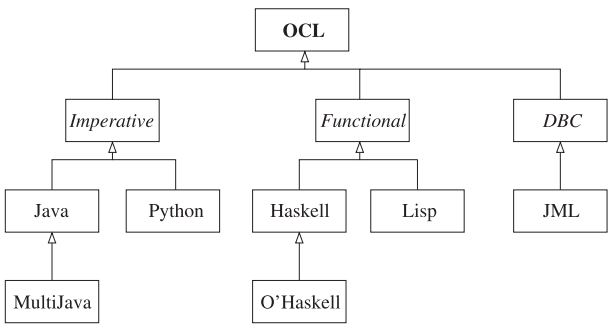


Fig. 4 Hierarchy of languages based on their structural similarities.

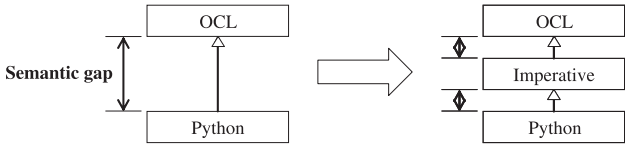


Fig. 5 Reducing semantic gaps.

By doing so, we can create one single framework for creating OCL assertion code generators for multiple languages, maximising reuse between different implementations. Also by minimising the semantic gap between the intermediary steps, we reduce the efforts required to understand OCL and the manual efforts of specifying the concept mapping from one step to another, as shown in Fig. 5.

3. Implementing Our Approach

In order to assess the feasibility of our approach, we have implemented *Rozyn*: an OCL translation tool based on the language similarities. Rozyn was implemented using the Maude System [9], a term rewriting system, which comprises powerful equational and rewriting logic capabilities, which would be useful for our multi-step translation.

3.1 Generation Process

As mentioned in the last section, we have two major steps of translation; (1) from OCL (precisely, an AST obtained by parsing an OCL description) to pseudo-code and (2) from pseudo-code to the target source code. The structure of pseudo-code depends on the class of the target languages. For example, we have a class of pseudo-code for imperative languages such as Java, C or Python, and have another class for functional programming languages such as Haskell or Lisp. We call this language of pseudo-code *Definition Language* (DL). The example in Fig. 3 (c) is the pseudo-code written in Imperative DL. The transformation then follows the hierarchy defined for the target language (see Fig. 4) starting at the top and proceeding to the leaf, performing translations for each intermediate step. Based on the hierarchy shown in Fig. 4, if we generate Python code from an OCL constraint, Rozyn first translates the constraint to Imperative DL, and then translates it to Python DL.

Figure 6 shows the overview of our process of generating an OCL assertion code from an arbitrary UML/OCL model, which consists of the following three main steps:

1. **UML + OCL → OCL AST**. The initial input to our system is a UML diagram annotated with OCL constraints. We therefore require means of interpreting the UML diagram and parsing the OCL constraints beforehand. After parsing a syntactically correct OCL description, it is then converted into its AST representation. UML, e.g., a class diagram, is used in the background to form an *environment* (ENV) containing the type information, which is stored along with the AST in a similar format. We use the Octopus tools [10] for parsing OCL constraints.
2. **OCL AST → Target language DL**. The target language for OCL constraints being executed is selected, and the hierarchy is traversed starting from OCL AST definition step-by-step, until the final output in the DL of the target language (target language DL) is produced. The translation is defined as rewriting rules in Maude and executed by using these rewriting rules and the type information. The detailed usage of the Maude will be mentioned later. This step is repeated until the target language DL, e.g., Python DL, is obtained.
3. **Target language DL → Assertion code**. Finally, the target language DL can be transformed into its equivalent executable code by applying a set of *printing rules*. We call this stage *printing*. The technique for this transformation is the same as the last step, i.e., we define the printing rules as rewriting rules in Maude.

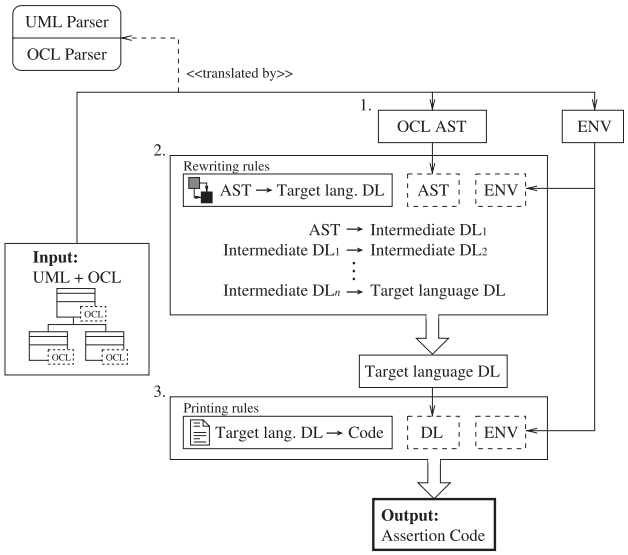


Fig. 6 Generation process.

Combined, these three steps represent a pluggable architecture, which could be inserted into an existing integrated development environment and used as a provider of assertion code bits fetched from the model specification. Such assertion code bits can be used in test cases or for runtime assertion.

For implementing Rozyn to automate the above generation process, as mentioned above, we have used the Maude language [9], which uses equational and rewriting logic. The language contains a functional-like data definition language used to define data structure and reduction rules, and a rewriting language used to describe rewriting rules between data structures. In the implementation of Rozyn, the data definition language was used to create each of the DLs, e.g., OCL AST (Fig. 9 (b)) or Imperative DL (Fig. 9 (d)), while the rewriting language was used for defining mappings between the different levels in the hierarchy, e.g., between OCL AST and Imperative DL.

Other implementation technologies, such as QVT-

based [11] meta-model based transformation engines, were also considered. However, we stopped on Maude because it was simple to use and provided minimum requirements such as tree-structure transformation functions.

3.2 Example of Transformation Process

Suppose you were given a simple model of a company shown in Fig. 7. If there was a company requirement that all employees must earn over 100,000, in OCL it could be expressed as an invariant on the `Company` class as shown at the bottom of the figure. This OCL constraint consists of a `forAll` expression, which enforces all `Employee` objects contained in the `employees` collection to have the `salary` property set to a value greater than `100000`. Generated Python assertion code from this OCL is shown in Fig. 8. The given OCL is transformed to the method `callExtMethod_A` and it is checked as the invariant via the function `invariant_1`.

As described in the architecture model in Fig. 6, the first step is to convert OCL expressions from the model into their AST representation. The OCL expression in our example can be expressed in the Maude language as shown in Fig. 9 (b), showing an example of mapping from OCL to its AST. In OCL `forAll` is a type of iterator expression, which we encode as `iteratorExp` AST node (Fig. 9 (b) line 1). The first argument is the collection to be iterated over, `employees` in our case (returned as the result of evaluating dotted block at lines 2 and 3 of Fig. 9 (b)). The second argument is the iterator expression type, `forAll` at line 4, followed by iterator variable and the sub-expression, at lines 5 and 6 through 8 (dotted block) respectively.

At the second step of Fig. 6, we enter the OCL AST rewriting stage where by means of rewriting rules we transform the OCL AST into the DL for a concrete implementation language. The intention of this step is to declare a transformation of the OCL concept into an abstract imperative code for its assertion. In our example, we declare a rewriting rule as shown in Fig. 9 (c). At the top of the figure, the left-hand-side of the rule (LHS) is declared to match all occurrences of `forAll`-expressions, on arbitrary collection expressions `OE`. Other variable parts of the matching rule are expressed in bold capitals. To transform the matched expression into Imperative DL, the right-hand-side (RHS) states that it should be expressed as an external method call (`impMethodExtract`) of return type `Boolean` that loops (`impForLoop`) through the target collection `OE` and tests (`impIf`) whether the sub-expression `OE'` holds. If sub-expression is not satisfied, the method returns (`impReturn`) with the boolean value *false* (`boolLitFalse`). The prefix `imp` identifies that the following structure belongs to the Imperative DL, and thus expressions such as *for*-loops, *if*-statements and *return*-statements are marked with that prefix. One possible implementation of this abstraction in Python is shown in Fig. 8.

When more context is required during translation, the environment (ENV) may be used to look up type information declared in the UML diagram. For example, `e.salary` in OCL is translated to `e.salary` in Python because the visibility of `salary` field was declared *public*. However, with tighter visibility `e.getSalary()` would be a correct translation.

By applying the above rule to the OCL AST in

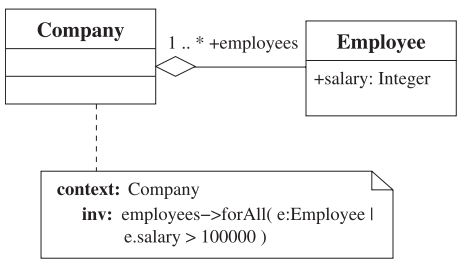


Fig. 7 An example model.

```
def callExtMethod_A(self):
    for e in self.employees:
        if (not(e.salary > 100000)):
            return False
    return True

def invariant_1():
    return self.callExtMethod_A()
```

Fig. 8 Generated Python code from the example input.

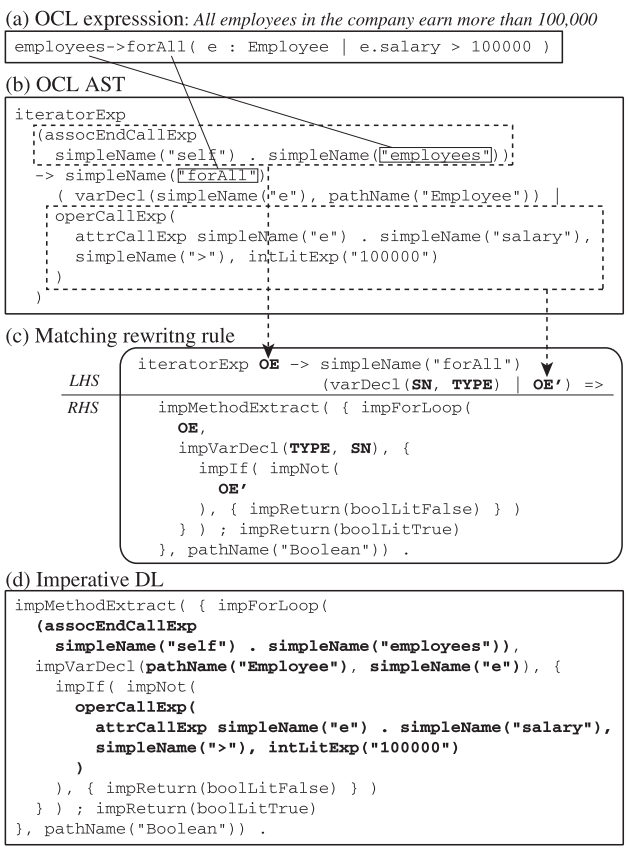


Fig. 9 An example of transformation from OCL to Imperative DL.

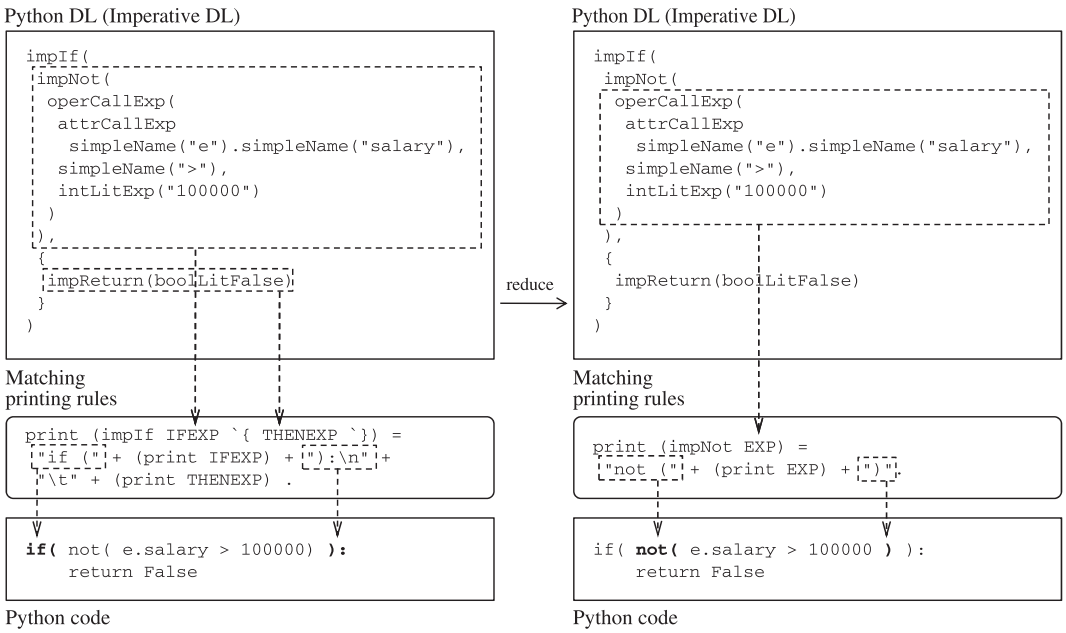


Fig. 10 Printing rules.

Fig. 9 (c), we obtain the imperative definition of the OCL constraint, shown in Fig. 9 (d). In this transformation, the iterator expression `e.salary > 100000` is mapped to the variable `OE'`, and as a result, appears inside the `impIf(impNot(···))` statement in the fifth line of Fig. 9 (d). The OCL constraint is now expressed in terms of the desired imperative language constructs.

As a rule, to produce executable Python code from the Imperative DL, it would first be transformed into Python DL in the same manner as the OCL AST was transformed into the Imperative DL. However, constructs of the target language that have very similar structure and semantics can be used directly without alternation from parent DL. This suggests that printing should also be done by directly referencing parent DL's constructs. Within the scope of our example, all generated Imperative DL constructs by coincidence have enough structural resemblance to Python, and thus we can apply printing rules to generate actual Python code directly from Imperative DL constructs (see bottom part of Fig. 10). In our example, Imperative DL is the most concrete DL to be used for printing, i.e., the leaf node.

Printing rules for the target language are matched and applied to the obtained DL top-down (Imperative DL in our case). An example of the printing process is depicted in Fig. 10, the external method call and the *for*-loop were omitted for the sake of brevity. The printing rule for *if*-block `impIf` is matched to the DL, and its sub-expressions, *if*-expression and *then*-expression are matched to variables `IFEXP` and `THENEXP` respectively. The printing rule states that in Python syntax *if*-block starts with `"if ("`, followed by the *if*-expression, then the closing bracket, a colon and then an indented *then*-expression. This reduction process is then repeated for each sub-expression (`impNot` on the right hand side of Fig. 10), until the whole DL is translated into Python code. For each target language a *printing module* is declared, containing such printing rules for all syntactical concepts of the language. Common syntactical rules can be expressed at higher levels of the hierarchy, for instance, the `impIf` rule in Fig. 10 is declared at the imperative level and therefore need not be declared explicitly at the Python level. Applying Python printing rules makes an executable Python code that can be plugged into a class implementing the `Company` and used to check the original OCL invariant.

3.3 Usage of Extending Hierarchy

For defining an OCL translator for a new language using our approach, it is generally enough to define a set (or modify an existing set) of printing rules to capture syntactical rules of that language, therefore detailed understanding of OCL will not be required. Modifications to output assertion code on syntactical level will always be reflected though changes to the printing rules. On the other hand, modifying structure will be done by changing rewriting rules higher up in the hierarchy e.g., Imperative DL (Fig. 9 (c)). As an example of such change, one could inject a logging action for whenever a *for-all* evaluation is made by adding `log("message");` appropriately in the RHS of the rewriting rule. This operation does not require direct understanding of how each concrete language implementation performs logging. If logging is not appropriate for some concrete language it can be omitted during next transformation or printing stage.

We can easily extend the language hierarchy shown in Fig. 4 by adding a new language DL. Translation of OCL concepts into concepts of other implementation languages requires a description of their transformation, for each concept. When adding of a DL is planned, a suitable parent DL is selected based on its similarity to the added DL. If no

```
subsorts ImpExp ExprBlock{ImpExp} < OclExpression .
...
sorts ImpIfExp .
subsorts ImpIfExp < IfExp .
subsorts ImpIfExp < ImpExp .
...
op impIf :
  OclExpression ExprBlock{ImpExp} -> ImpIfExp [ctor] .
...
```

Fig. 11 An example of structure declarations.

similar parent DL exists, the root language, i.e., OCL AST, can be used for directly translating OCL to the target DL. Next, we write *structure declarations* defining the new concepts of the target DL and how they relate with the concepts of the parent DL. For instance, the example of Maude structure declarations shown in Fig. 11 includes at least five data definitions related to *if*-expressions in Imperative DL. Finally, we write transformation rules from the parent to the target DL and printing rules such as the examples in Figs. 9 and 10.

3.4 Limitations and Disadvantages

During the implementation of our approach, we have discovered some limitations. Here we have a list, which is by no means complete, but outlines the most obvious limitations.

Insertion in the middle. Even though the existence of the language hierarchy allows us to easily append new languages, it has the disadvantage of being difficult to extend from the middle, i.e., regrouping existing languages and extracting a common DL would create a new node in the middle of the hierarchy. Such operation would be difficult because all of the child languages of the new DL would have to be modified (expressed in terms of the new pseudo-language).

Hybrid languages. We have also not provided any means for making it possible to declare DLs with more than one parent DL. This could be useful with some hybrid languages that inherit common properties from different language types. In our approach, we traverse the hierarchy from top to bottom, taking a single path, thus it is not currently possible for DLs to have multiple parent DLs.

4. Experimental Evaluation

To evaluate our approach, we made an experiment. The aim of the experiment is to show that our approach gives us the ability to flexibly create OCL assertion code generators for multiple languages, gaining savings in manual efforts required to implement each generator.

Evaluation assumes that users of our tool deal with projects what would benefit from automated OCL translation. As an example of such project, there is work by Chimiak-Opoka [12] describing a large scale application with thousands of lines of OCL code. For this reason, completely manual translation of OCL is not included in evaluation.

4.1 Procedure

For evaluation, we will implement four OCL translators for four different languages. Two languages are from under the imperative languages hierarchy, Java and Python; and the others are from under the functional languages hierarchy, Haskell and O'Haskell. Then the efforts required to implement each one of those OCL translators will be measured and compared to an estimated effort required to create an OCL translator for the same language using a direct approach, i.e., the development from scratch. The direct approach assumes an implementation of OCL requiring minimum effort, realised by simply implementing a set of transformation rules that directly translate each OCL expression into the target language. Finally, generated assertion code for each OCL translator will be checked to make sure they exhibit desired behaviour.

We first build an OCL translator for Python, using the direct approach, and use the resulting effort figure as a yardstick for estimating direct approach efforts for other languages. Secondly, we pre-build the Imperative and the Functional DLs, and then implement OCL translators for each one of the four languages by extending the Imperative or the Functional DLs as appropriate, and measure the efforts during each implementation. Lastly, the efforts that were required to create each OCL translator as an extension in our approach are compared to the efforts for the direct approach.

To evaluate the effort at each step, we need a comparable indicator that can be used to measure and compare these efforts. In order to evaluate the effort for creating OCL translators, we will count the following evaluation parameters:

1. the number of rule definitions in the rewriting language, e.g., in Fig. 9 (c) (*RR*),
2. the number of structural declarations in the data definition language, e.g., in Fig. 11 (*S*), and
3. the number of definitions in the printing module, e.g., shown in Fig. 10 (*PR*)

in Maude, for each generator. For evaluation the total of the above parameters will be compared with the number of rewriting rules required to create an OCL assertion code generator without the use of the intermediate definition languages i.e., the *direct* approach.

Note that the main feature of our approach is in the fact that creating a generator for a new language only requires the developer to provide details specific to the language in its category. For example, creating a generator for Java would require to provide structural description and syntax for Java class cast expressions as they will be used in the *foreach*-loops and possibly other Java constructs, but not necessarily in other imperative languages, such as Python. For this reason, the evaluation parameters described only need to be counted when creating the final node (the leaf) in our hierar-

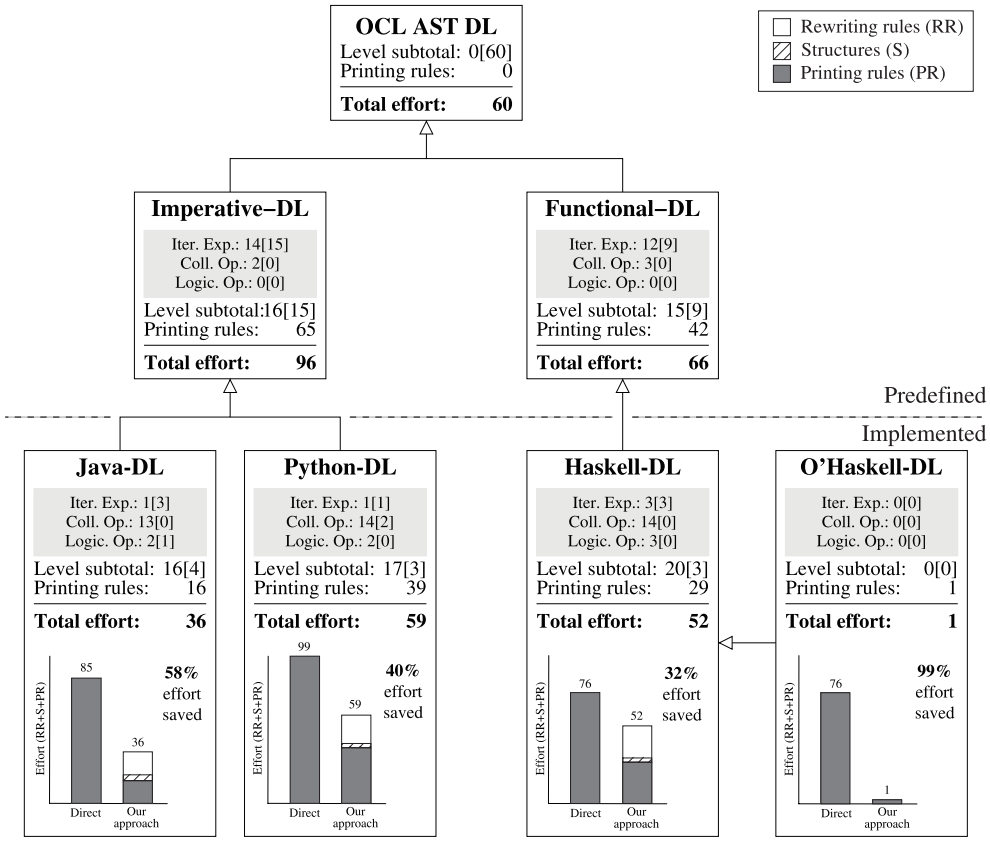


Fig. 12 Evaluation results.

Table 1 Evaluation targets.

Iterator Expressions (Iter. Exp.)		Collection Operations (Coll. Op.)		Logical Operators (Logic. Op.)
iterate	select	includes	isEmpty	xor
forAll	reject	excludes	notEmpty	implies
exists	any	includesAll	union	
isUnique	one	excludesAll	intersection	
collect	sortedBy	including	flatten	
		excluding	sum	
		first	size	
		last	at	

chy, since we can assume that parent nodes are predefined.

The complete list of all implemented OCL features is given in Table 1. All features are subdivided into three main groups: iterator expressions, collection operations and logical operations. Finally, the generated assertion code for each OCL generator was manually checked to make sure it exhibits desired behaviour, using some test cases. At the current stage, the focus was put on evaluating the efficiency of our approach; therefore the correctness of generated assertion code is responsibility of the developers of rewriting rules.

4.2 Results

In order to clearly show the efforts saved using our approach compared to implementing directly, we have summarised our results in Fig. 12. The numbers are given in format *RR*[*S*] showing the number of rewriting rules and number of structures defined, subdivided according to groups in Table 1. In the example of Java, the numbers of rewriting rules and of structures for iterative expressions are 1 and 3 respectively. Note that O'Haskell is an extension of the Haskell comprising several behavioural and syntactical changes. However, implementation of OCL assertion code generator only required a modification in one syntactical rule in the printing module (see bottom right of Fig. 12).

For each target language, we have shown the estimated effort of direct implementation (bar on the left) and actual effort by our approach (bar on the right). In the example of Java, 36 and 85 rules were written in our approach and direct one respectively. We have also shown the percentage of the effort saved in case when our approach is undertaken. In Java, we could reduce 49 (85 – 36) rules and as a result, 58% (49/85) of the effort could be reduced. Each bar-chart assumes that the parent node in the graph is predefined.

4.3 Discussions

From Fig. 12 we can see that on average we are saving approximately 50% effort, which clearly indicates that languages share a fair amount of structural and semantic similarities and reusing those similarities is very efficient. The savings in effort that can be seen from the results are a good

indication that languages in the same structural family can be related also on the semantic level and hence can be effectively grouped for reuse. It was also shown that small changes in the syntax of the language require proportionally small efforts of implementation.

In addition, as can be seen from Fig. 12, most of the complex OCL iterator expressions (Iter. Exp.) could be rewritten using non-OCL concepts, in other words concepts from Imperative DL or Functional DL, which are independent of OCL. This underlines the fact that our approach alleviates the need to understand OCL completely.

Measuring effort involved in implementing an OCL translator in an unbiased manner was a very difficult task. In our effort evaluation strategy, we have tried to cover most complexity aspects associated with implementation of OCL translators, by quantitatively measuring the number of rewriting rules, printing rules and semantic structures. However, implementation of some rewriting rules was more complex than others, and not because of their size, but because of their dependency on other rewriting rules (i.e., application of such rule must be followed by application of another rule). Even though we have tried to take into account the complexity of rewriting rules by counting the number of structures defined to support them, it was not always a complete indication of effort. On the other hand, some rewriting rules were very easy to specify, because they were simply representations of a concept in the target language, such as a *for-loop*, and were not directly related to OCL.

Representativeness of evaluation results is not ideal as all evaluation procedures were carried out by the same person. For a fairer result evaluation with adding new languages by several persons should be considered.

5. Related Work

Currently available approaches for evaluating OCL constraints can be split into three main types: metamodel-based model validation, source code assertion and translating OCL to another Design-by-Contract (DBC) language.

Metamodel-based model validation. Checking for correctness of OCL constraints for a model using its metamodel description can be advantageous since such approach can take arbitrary data models as their input, and therefore in theory it is possible to validate absolutely any type of data. Some researches that use this approach include Kent OCL Tool [6], NAOMI [7], and ITP/OCL [8]. However, very few language platforms provide direct inspection of objects at run-time (also known as reflection) which would be necessary for validation. Also, further knowledge of the underlying reflection API would be necessary.

Checking constraints on the implementation level. Source code assertion approaches usually use code instrumentation or aspect-oriented techniques to achieve code checking at run-time. However, all such approaches, including jContractor [13], Handshake [14], ocl2j [15], Jass [16], and iContract [17], are tailor-made for a specific programming language.

Translating OCL to JML. Another approach is to translate OCL to another OCL-like DBC language such as JML [18]–[20]. Hamie has proposed a set of mappings from OCL to JML [21] to which we have previously contributed with our own extensions [22]. However, the problem with such approaches is that there is currently no other DBC language that can be applied to multiple programming languages.

In informatics, introducing abstraction layers for minimising costs of software development activities is one of the orthodox approaches. For example, some compilers such as GNU Compiler Collection (GCC) [23] use a similar technique to translate various front-end language such as C, C++ or Java, into an intermediate representation, which can then be optimised and translated into the target platform machine code. The main difference between GCC and our approach is that, regardless of the target platform, there is only one intermediate representation format (but with possibly different optimisations applied).

6. Conclusion

To conclude, we first focused on the problem of working with OCL constraints on the implementation level for multiple languages. We also proposed our approach to remedy this problem and performed an experiment to confirm the claimed effort savings when using our approach. We showed how new OCL translators can be added without knowledge of OCL and how functionalities such as logging can be easily injected into OCL translator implementations. Our experimental evaluation with Rozyn shows the effectiveness of our approach.

7. Future Work

Some OCL functionality was not covered, such as history expressions, OCL messages and the `allInstances` call, that we have not implemented and left for future work.

In order to merge the code generated from UML and assertion code generated by Rozyn, we could develop a plug-in for an integrated development environment, such as Eclipse. This aspect is considered for future work.

To manage the framework code, including language structures, rewriting and printing rules, we will need to develop a management scheme. The scheme could consist of guidelines and conventions, to help keep code consistent across the framework, and allow collaboration of multiple developers.

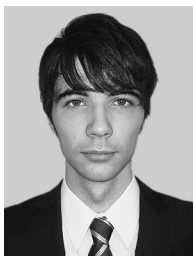
With regard to evaluation of our approach, we realise the need to consider the following in the future:

- Evaluation of effort involved in learning the DL used for rewriting,
- Evaluation with several persons and greater variety of languages (including hybrid languages), and
- Confirming that generated assertion code exhibits consistency, accuracy and determinateness, as proposed by Gogolla et al. [24].

We have also realised of certain limitations of our approach, such as difficulty in introducing DLs into the middle of the hierarchy because this would cause change to propagate to all nodes below and would be difficult to automate. Catering for this is a significant task that will require an individual approach in the future.

References

[1] R. Moiseev, S. Hayashi, and M. Saeki, "Generating assertion code from OCL: A transformational approach based on similarities of implementation languages," Proc. 12th International Conference on Model Driven Engineering Languages and Systems (MoDELS 2009), pp.650–664, 2009.
[2] OMG, "Model-Driven Architecture," available at http://www.omg.org/mda/
[3] OMG, "Object constraint language specification, version 2.0," available at http://www.omg.org/technology/documents/formal/ocl.htm
[4] Omondo, "EclipseUML," available at http://www.omondo.com/
[5] G. Kiczales, J. Irwin, J. Lamping, J.M. Loingtier, C.V. Lopes, C. Maeda, and A. Mendhekar, "Aspect-oriented programming," Proc. 11th European Conference on Object-Oriented Programming (ECOOP 1997), pp.220–242, 1997.
[6] D.H. Akehurst and O. Patrascoiu, "OCL 2.0 – Implementing the standard for multiple metamodels," Electronic Notes in Theoretical Computer Science, vol.102, pp.21–41, 2004.
[7] F. Chabarek, "Development of an OCL-parser for UML-extensions," Master's thesis, Technical University of Berlin, 2004.
[8] M. Clavel and M. Egea, "ITP/OCL: A rewriting-based validation tool for UML+OCL static class diagrams," Proc. 11th International Conference on Algebraic Methodology and Software Technology (AMAST 2006), pp.368–373, 2006.
[9] The Maude Team, "The Maude System," available at http://maude.cs.uiuc.edu/
[10] A.K. Jos Warmer, "Octopus: OCL Tool for precise UML specifications," available at http://octopus.sourceforge.net/
[11] OMG, "Meta Object Facility (MOF) 2.0 Query/View/Transformation (QVT)," available at http://www.omg.org/spec/QVT/
[12] J. Chimiak-Opoka, "OCLLib, OCLUnit, OCLDoc: Pragmatic extensions for the object constraint language," Proc. 12th International Conference on Model Driven Engineering Languages and Systems (MoDELS 2009), pp.665–669, 2009.
[13] M. Karaorman, U. Hölzle, and J.L. Bruno, "jContractor: A reflective Java library to support design by contract," Proc. 2nd International Conference on Meta-Level Architectures and Reflection (Reflection 1999), pp.175–196, 1999.
[14] A. Duncan and U. Hölzle, "Adding contracts to Java with Handshake," Tech. Rep. TRCS98–32, Department of Computer Science, University of California, 1998.
[15] W.J. Dzidek, L.C. Briand, and Y. Labiche, "Lessons learned from developing a dynamic OCL constraint enforcement tool for Java," Proc. Workshop on Tool Support for OCL and Related Formalisms – Needs and Trends, pp.10–19, 2005.
[16] D. Bartetzko, C. Fischer, M. Möller, and H. Wehrheim, "Jass – Java with assertions," Electronic Notes in Theoretical Computer Science, vol.55, no.2, pp.1–15, 2001.
[17] R. Kramer, "iContract – The Java design by contract tool," Proc. Technology of Object-Oriented Languages and Systems (TOOLS 1998), pp.295–307, 1998.
[18] L. Burdy, Y. Cheon, D. Cok, M. Ernst, J. Kiniry, G.T. Leavens, K.R.M. Leino, and E. Poll, "An overview of JML tools and applications," International Journal on Software Tools for Technology Transfer, vol.7, no.3, pp.212–232, 2005.
[19] Y. Cheon and G.T. Leavens, "A runtime assertion checker for the Java Modeling Language (JML)," Proc. International Conference on Software Engineering Research and Practice (SERP 2002), pp.322–328, 2002.
[20] G.T. Leavens, K.R.M. Leino, E. Poll, C. Ruby, and B. Jacobs, "JML: Notations and tools supporting detailed design in Java," Companion Proc. 21st International Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA 2000), pp.105–106, 2000.
[21] A. Hamie, "Translating the object constraint language into the Java modelling language," Proc. 2004 ACM Symposium on Applied Computing (SAC 2004), pp.1531–1535, 2004.
[22] R. Moiseev and A. Russo, "Implementing an OCL to JML translation tool," IEICE Technical Report, SS2006-58, 2006.
[23] Free Software Foundation, Inc., "GCC, the GNU compiler collection," available at http://gcc.gnu.org/
[24] M. Gogolla, M. Kuhlmann, and F. Büttner, "Benchmark for OCL engine accuracy, determinateness, and efficiency," Proc. 11th International Conference on Model Driven Engineering Languages and Systems (MoDELS 2008), pp.446–459, 2008.



Rodion Moiseev received a B.Eng. degree in computer science from Imperial College of London in 2005. He also recieved a M.Eng. degree in computer science from Tokyo Institute of Technology in 2007. He is currently working at Infoscience Corporation. His research interests include model driven engineering, software design, and programming languages.

Shinpei Hayashi received a B.Eng. degree in information engineering from Hokkaido University in 2004. He also recieved M.Eng. and Dr.Eng. degrees in computer science from Tokyo Institute of Technology in 2006 and 2008, respectively. He is currently an assistant professor of computer science at Tokyo Institute of Technology. His research interests include software changes, software patterns, software development environment, and program understanding.

Motoshi Saeki received a B.Eng. degree in electrical and electronic engineering, and M.Eng. and Dr.Eng. degrees in computer science from Tokyo Institute of Technology, in 1978, 1980, and 1983, respectively. He is currently a professor of computer science at Tokyo Institute of Technology. His research interests include requirements engineering, software design methods, software process modeling, and computer supported cooperative work (CSCW).