# Practical Application of a Translation Tool from UML/OCL to Java Skeleton with JML Annotation

Kentaro Hanada[1], Kozo Okano[1], Shinji Kusumoto[1] and Kiyoyuki Miyazawa[1]
[1]*Graduate School of Information Science and Technology, Osaka University, Japan*
{*k-hanada, okano, kusumoto, k-miyazw*}*@ist.osaka-u.ac.jp*

Keywords: Model-Driven Architecture, OCL, JML, design by contract

Abstract: In recent years, MDA techniques have been strongly developed. Thus, translation techniques such as UML to some program languages have gained a lot of attention. Translation techniques such as OCL to JML have been also researched. OCL is a language to describe detail properties of UML and standardized by OMG, while, JML is a language to specify properties of Java program. Both OCL and JML are based on DbC and able to provide properties of classes or methods. There are, however, not many researches on translating automatically OCL into JML and past researches often pay little attention to collection features, especially iteration. Our research group has proposed a concrete method which translates UML class diagram with OCL into Java skeleton with JML. This paper presents an implementation tool based on the technique. To evaluate the quality of the tool, we applied the tool to two real examples, a warehouse management program and a syllabus management system. As a result, we found that every OCL constraint described manually was translated successfully into JML. Also, we found some defects existed in the design of a syllabus management system.

## 1 Introduction

In recent years, MDA (Model Driven Architecture) (Kleppe et al., 2003) techniques have been strongly developed. Thus, translation techniques such as UML to some program languages have gained a lot of attention.

Translation techniques such as OCL (Object Management Group, 2006) to JML (Leavens et al., 1999) have been also studied. OCL is a language to describe detailed properties of UML and standardized by OMG (Object Management Group), while, JML is a language to specify properties of a Java program. JML aims for describing more detail properties than OCL does. Both OCL and JML are based on DbC (Design by Contract) (Meyer, 1992) and able to provide properties of classes or methods. There are not, however, many researches on translating automatically OCL into JML, and past researches often pay little attention to collection features, especially iteration. The problem should be solved because the iteration feature is necessary and widely used.

We have already proposed a concrete method which translates a UML class diagram with OCL into a Java skeleton with JML (Owashi et al., 2010). This paper presents a tool implementing the method. The tool is applied to two real examples, a warehouse management program and a syllabus management system, which consist of seven classes and 60 classes, respectively. As a result, we found that every OCL constraint described manually was translated successfully into JML. In addition, 86% of them are translated into the same form of JML described manually and independently. The translation time is less than seven seconds for the syllabus management system. Consequently we can conclude that our tool can practically translate OCL statements.

The contributions to the field are the following three items. The first one is implementation of the translation tool as an Eclipse plug-in. The tool supports iteration feature that is not supported by a lot of existing research of translation from OCL to JML. The second one is that the proposal method utilizes external libraries (including Java standard libraries and third party's libraries) to improve the utility of the translation tool. The last one is an evaluation of the translation tool. We carried out evaluation experiment onto real projects.

The organization of the rest of the paper is as follows. Sec.2 describes the background of this research and related work. Sec.3 presents overview of our tool. Sec.4 and Sec.5 will give experimental results and

discussions, respectively. Finally, Sec.6 concludes the paper.

## 2 Background

In this section, we present background of our research such as some techniques and related works.

### 2.1 OCL and JML

OCL details properties of UML models. It is standardized by OMG. UML diagram alone cannot express rich semantics of and all relevant information about the target design. OCL allows describing precisely the additional constraints on the objects and entities in a UML model.

JML is a language to detail constraints of Java methods or objects (Leavens et al., 1999). The constraints are based on DbC. It is easy for novices to describe properties in JML because the syntax of JML is similar to that of Java.

### 2.2 Related Work

Hamie (Hamie, 2004) proposed a method which translates from OCL into JML based on syntax conversion techniques. Rodion and Alessandra (Rodion and Alessandra, 2006) enhanced the technique of Hamie (Hamie, 2004). They proposed translation techniques of Tuple type operations and a part of Collection type operations. Moreover, they implemented these techniques. Specifically, OCL operations that do not correspond operations of JML or Java directly such as $setOfSets->flatten()$ were solved by defining generalized libraries like expression (1)

$$\text{JMLTools.flatten}(setOfSets) \tag{1}$$

Avila et al. (Avila et al., 2008) proposed a method which reduces differences of collection operations among OCL and JML. They also provide a Java class library which interprets some of OCL statements. These methods, however, do not enough support an iterate feature that is the most basic operation among collection loop operations. An iterate feature is an operation which applies an expression given as its argument to each element of a collection which is also given as its another argument.

$$\begin{aligned}\text{Set}\{1,2,3\}->\ &\text{iterate}(i\text{: Integer};\\ &sum : \text{Interger} = 0 \mid sum + i)\end{aligned} \tag{2}$$

Expression (2) defines an operation that returns a value which represents a sum of all elements in Set.

```
private T1 mPrivateUseForJML01(){
   μ(init);
   for (T2 e: μ(c1))
      res = μ(body)
   return res;
}
```

Figure 1: General Java Template of the Method for Iterate Feature

In expression (2), the first argument ($i : Integer$) defines an iterator variable. The second argument ($sum : Integer = 0$) defines a variable which is used to store the return value and its initialization. The third argument ($sum + i$) stands for an expression that is executed iteratively in the loop.

Our research group proposed a technique to resolve this problem by inserting a Java method that is semantically equal to each OCL loop feature (Owashi et al., 2010). It is worthful that the algorithm deals with the iterate feature because an iterate feature is widely used.

Expression (3) shows the general format of an iterate feature. The variables *e*, *init*, *body* and *c* mean an iterator variable, a declaration of the return value and its initialization, an expression executed in the loop, and a Collection type variable respectively.

$$c-> \quad \text{iterate}(e; init \mid body) \tag{3}$$

Figure 1 shows a general format of our newly created method. The keywords $\mu()$, $T_1$, $T_2$ and the variable *res* mean a function which translates an OCL expression into a Java expression, a variable declared in *init*, a variable *e*, and the name of a variable declared in *init* respectively.

## 3 Implementation

In this section, we will present the implementation of our translation tool. Figure 2 shows a brief overview of our translation tool. We have implemented the tool as an Eclipse plug-in. In this section, firstly, we show the detailed explanation about the implementation of the parser of OCL. Secondly, we explain in detail about how to support the external type information. Finally, we list parts of translation rules from OCL statements to JML statements.

### 3.1 Translation from UML to Java

We utilize an existing resource to translate from UML to Java. Papyrus UML (Eclipse Foundation, 2012a) is used as roles of drawing UML and translation from

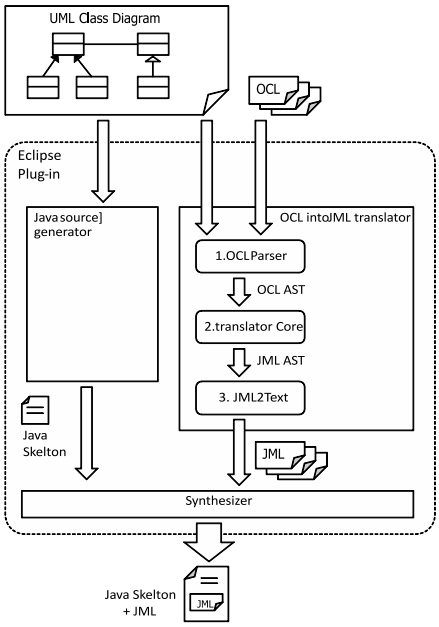


Figure 2: Brief overview of our translation tool

UML to Java. Papyrus UML is an open-source UML CASE tool. It is developed as one of Model Development Tools (MDT) projects in Eclipse Foundation. It is available as an Eclipse plug-in. Its input and output files are compliant with XMI, the standard UML file format by OMG.

## 3.2 Implementation of OCL Parser

Firstly, we describe the parser generator. We use ANTLR (Parr, 2007) for the parser generator. ANTLR automatically generates a parser using LL($k$) parsing. ANTLR supports Java as an output language. It generates not only a parser but also its associate lexical analyzer. Paper (Object Management Group, 2006) defines EBNF of OCL. It contains some modification in EBNF. Therefore, we modify the EBNF. As a result, the grammar of OCL statements is constructed by EBNF with the size of 50 non-terminal symbols and 100 generation rules.

Secondly, we describe a method to support for external libraries. We designed the OCL parser to parse only two kinds of types. One is a defined type by a user. The other is a standard type in OCL. It is, however, difficult to design without using the standard library of Java and external packages developed by third parties. For this reason, the translation tool needs to use information on types of external libraries and the standard library. There are several approaches. One approach is that the translation tool is applied to UML class diagrams with extra diagrams that have enough information on standard library and third parties libraries. These extra diagrams are prepared by a user in advance. The other approach is that the OCL parser accepts directly information on such extra types. There are, however, vast amounts of classes in the standard library. Also, flexible acceptance of external libraries seems difficult. For these reasons, we choose the first approach. In order to obtain UML class diagrams with extra diagrams, we use Java2UML (Atos Origin TOPCASED Team, 2006). The input is a Java project or a package folder. The output is UML diagrams corresponding to the Java files in the input directory. We improved Java2UML, so that it can read a single Java file specified in the input. It also obtains super classes of the input class. From the information, a user can specify a subset of super classes that are needed for further analysis.

Finally, we describe translation rules of OCL statement to JML statement. The set of the translation rules is defined in Owashi (Owashi et al., 2010). We, however, defined new translation rules due to the set of the translation rules are not sufficient. Table 1 is a set of new translation rules. The number of whole rules is about 150.

A translation function of an OCL statement to a JML statement is expressed by $\mu$ in the same manner as (Hamie, 2004). Any Integer, Real and Boolean are expressed by $a_i$. Any Collection is expressed by $c_i$.

# 4 Experiments

This section will explain experiments in detail. We conducted two experiments. The first experiment uses a small project (Experiment 1). The second experiment (Experiment 2) is larger than Experiment 1.

## 4.1 Preparation of Experiment

We need OCL statements and ideal JML statements for reference. OCL statements are used to generate JML statements by our translation tool. Ideal JML statements are used to compare and analyze generated JML statements by our translation tool as references. The project of Experiment 1, however, does not have OCL statements. In order to perform Experiment 1, we described OCL statements. The OCL statements

Table 1: new $\mu$ translation

| | | |
|---|---|---|
| $a_1$.oclIsKindOf($a_2$) | = | $\mu(a_2)$.class.isAssignableFrom ($\mu(a_1)$.getClass()) |
| $a_1$.oclIsTypeOf($a_2$) | = | $\mu(a_1)$.getClass().equals($\mu(a_2)$) |
| $a$.oclIsUndefined() | = | $\mu(a)$ == null |

are syntactically identical and logically equivalent to ideal JML statements.

The project used in Experiment 2 does not have not only OCL statements but also JML statements. We, however, need both data in order to perform an Experiment 2. Therefore, firstly, we described JML statements on the basis of the specification which is read from the codes and comments of javadoc. Secondly, we checked the correctness of the JML statements by running test cases with a runtime assertion checker, JMLrac. Finally, we described OCL statements that are syntactically identical and logically equivalent to ideal JML statements.

In order to evaluate the quality of JML statements generated by the translation tool, we measured three items, coverage, recall ratio, and execution time of the runtime assertion checker.

Here, firstly we provide a metric for the coverage. We define the coverage by Expression (4), where $C_{all}$ and $C_{translatable}$ are the number of pre-conditions and post-conditions, and the number of statements translated by the translation tool, respectively.

$$Coverage = C_{translatable}/C_{all} \quad (4)$$

Secondly, we provide the recall ratio. We define the recall ratio by Expression (5), where $C_{equal}$ is the number of statements which are translated by the translation tool and are identical to the corresponding ideal JML statement. Here, "corresponding" stands for syntactical and logical equivalence.

$$recallratio = C_{equal}/C_{translatable} \quad (5)$$

Finally, we compare execution times of runtime assertion checker, JMLrac by applying source codes with JML statements generated by the translation tool and that with ideal JML statements. We use JML4c (Sarcar and Cheon, 2010) and JML4rt (Sarcar and Cheon, 2010) as a JML compiler and a runtime assertion checker, respectively. As far as we know, JML4c is the only compiler that supports enhanced for-loop and generics. Also for a given executable file generated by JML4c, JML4rt checks consistency between runtime values of variables and the constraints given by the JML statements. When some of constraints of JML statements violate to the runtime values of variables, JML4rt outputs the location of violated point and the values of variables at the location.

## 4.2 Environment of Experiments

Evaluations have been performed on an Intel(R) Core(TM)2 Duo E7300 2.66GHz 2.67GHz HP Compaq dx7500 Microtower with 4GB RAM running Windows Vista 64bit.

## 4.3 Experiment 1

In Experiment 1, we evaluated the quality of the translation using the warehouse management program. It consists of seven classes. Table 2 shows components of the warehouse management program in details. It (Owashi et al., 2008) has correct JML statements by the past research. In Experiment 1, we add OCL statements for 20 methods.

In Experiment 1, as we described, JML statements have already been fully annotated. We selected three core classes and annotated OCL statements. For the three classes, we can translate every OCL statements into correct JML statements. Consequently, Coverage and recall ratio are both 100 %, which indicates the power of the tool. Also, we observed execution times of both cases. The results show that there is no distinguishing execution time of manual JML statements from that of JML statements by the translation tool.

Execution time of translation from OCL statements to JML statement is 20ms. Also, execution time of translation from abstract tree of OCL to abstract tree of JML is 1ms. Therefore, if we describe OCL statements to all attributes and all methods of the program, perhaps execution time of translation from these OCL statements (about one hundred statements) to JML statement will be about 2 sec.

## 4.4 Experiment 2

In Experiment 2, we evaluated the quality of translation tool by data of educational material of IT-Spiral (Ministry of Education, Culture, Sports, Science and Technology, 2010). The project is a syllabus management system of Wakayama University. The system consists of 200 classes, and we choose core 60 classes and annotate 400 methods in the chosen classes. Table 3 shows parts of OCL statements and JML statements.

Table 4 shows the result of translation from OCL statements of the syllabus management system to

Table 2: Components of warehouse management program

| Class Name | # of methods | # of lines |
|---|---|---|
| ContainerItem | 12 | 224 |
| Customer | 10 | 156 |
| Item | 7 | 110 |
| ReceptionDesk | 8 | 162 |
| Request | 16 | 245 |
| StockState | 0 | 9 |
| Storage | 10 | 258 |
| TOTAL | 63 | 1164 |

```
/*@ requires this.getDAO(UserDAO.class)
                                        != null;@*/
public void save(final User user)
                        throws ServiceException{
    getDAO(UserDAO.class).save(user);
}
```

Figure 3: Save method of UserServiceImpl class

JML statements. It shows that the coverage and the recall ratio are 89.9% and 86.0%, respectively. All OCL statements are translated to JML statements. The JML statement that is not identical to ideal JML statement is constraints with iteration feature.

Half of JML statements are not translated due to following two cases.

1. JML statements including class literal
2. JML statements including an array expression

For example, JML statement includes class literal shown in Figure 3. We could not describe the statement as OCL statements. We described "requires this.getDAO(UserDAO.class) != null;" as JML in the save method because the method must check that DAO is not null as pre-condition. UserDAO class is, however, the class literal; we could not describe the corresponding OCL statements.

Figure 4 shows the example statements including arrays. The post-condition checks that values in this.kyoukasho have the same values in the array of parameter. OCL, however, uses Sequence instead of the array; we cannot describe the corresponding OCL statements. We will explain the details in Section 5.

# 5 Discussions

## 5.1 Some Observations on the Results

The translation tool can translate 90 percent out of all OCL statements in 60 classes into JML statements. We, however, could not translate the following JML

Table 3: The number of classes and methods which are annotated in OCL

| Class | # of classes | # of methods |
|---|---|---|
| ServiceImpl | 13 | 74 |
| DAO | 11 | 64 |
| Util | 6 | 29 |
| DTO | 2 | 25 |
| Entity | 30 | 216 |
| TOTAL | 62 | 408 |

```
/*@
ensures kyoukasho != null ?
  (\forall int i4; i4>=0
  && i4<this.kyoukasho.length;
  kyoukasho[i4] == null ||
  this.kyoukasho[i4].equals(kyoukasho[i4]))
  && this.kyoukasho.length == kyoukasho.length
  : this.kyoukasho == null;
@*/
public void setKyoukasho
   (final JugyouShousaiKyoukasho[] kyoukasho){
    this.kyoukasho = kyoukasho;
}
```

Figure 4: JML statements of the method which contains arrays

statements to OCL statements: assignable keyword, class literal, and arrays.

There is not a keyword corresponding to "assignable" in OCL. Also, there is not concept of class literal in OCL because class literal is a proper concept in Java language. OCL does not define arrays, but Sequence type is provided to represent a list structure. Sequence is an indexed bag (multi-element set). It supports insertAt() and prepend() features, thus Sequence can be translated into List in Java. Our tool also translates in the above manner. On the other hand, there are few demands to translate sequence feature into an array. It is, however, arguable that it supports translation from sequences to multi dimension arrays for some reasons such as efficiency.

## 5.2 Threats to Validity

Here, we simply summarize threats to validity. As external threats to validity, we can say the number of experiments is not enough. Therefore, we cannot generalize the results and discussion. We, however, choose real examples; the tendency can be inferred. Of course, to obtain more precise data, we have to apply the tool to more applications.

As internal threats to validity, we can enumerate

Table 4: The result of Experiment 2 to the syllabus management system

| | |
|---|---|
| The number of JML statements by hand($C_{all}$) | 602 |
| The number of OCL statements | 541 |
| The number of translated JML statements by the tool($C_{translatable}$) | 541 |
| The number of match of translated JML statements and ideal JML statements($C_{equal}$) | 465 |
| The number of match by inserting methods | 57 |

(1) that we have constructed OCL statements manually from JML statements and (2) that we do not translate all OCL statement of all class in both Experiments.

As for (1), if OCL statements are described by a person who knows about the detail of translation algorithm of out tool, it may be said that the OCL statements include a problem in our experiments. OCL statements, however, are described manually by a person who does not know about the detail of our translation tool. We think this shows the validity of the OCL statements in our experiments.

As for (2), we choose core classes of both cases. For example, Experiment 2 uses core classes that are used to access its database. We analyzed syntactically all patterns that are covered in the description. Also, the main objective of the experiments is how many OCL statements are translated into JML statements. We think it is not serious problem that we cannot annotate to every class.

## 6 Conclusion

This paper presents a tool which translates OCL including iteration features into JML. On implementation of the tool, we enhance our tool to be able to deal with external libraries that are not included in design projects such as Java standard library. We applied our tool to two real examples, a warehouse management program and a syllabus management system. As a result, we found that 90% of necessary constraints were well translated, and 86% of constraints were translated into suitable expressions. The translation time in which about 600 OCL expressions are translated is less than seven seconds. Consequently we can conclude that our tool can translate in practical time.

Future work includes reverse translation from JML to OCL. Implementing mutual transformation between OCL and JML by use of Xtext (Eclipse Foundation, 2012b) is considerable as future work.

## 7 Acknowledgments

This work is being conducted as Grant-in-Aid for Scientific Research C (21500036).

## REFERENCES

Atos Origin TOPCASED Team (2006). Java to UML. http://gforge.enseeiht.fr/projects/java2uml/.

Avila, C., Flores, Jr., G., and Cheon, Y. (2008). A Library-Based Approach to Translating OCL Constraints to JML Assersions for Runtime Checking. In *International Conference on Softw. Eng. Research and Practice*, pages 403–408.

Eclipse Foundation (2012a). Papyrus UML. http://www.eclipse.org/modeling/mdt/papyrus/.

Eclipse Foundation (2012b). Xtext - Language Development Framework. http://www.eclipse.org/Xtext/.

Hamie, A. (2004). Translating the Object Constraint Language into the Modeling Language. In *In Proc. of the 2004 ACM symposium on Applied computing*, pages 1531–1535.

Kleppe, A., Warmer, J., and Bast, W. (2003). *MDA explained: the model driven architecture: practice and promise*. Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA.

Leavens, G., Baker, A., and Ruby, C. (1999). JML: A Notation for Detailed Design. *Behavioral Specifications of Businesses and Systems*, pages 175–188.

Meyer, B. (1992). *Eiffel: the language*. Prentice-Hall, Inc., Upper Saddle River, NJ.

Ministry of Education, Culture, Sports, Science and Technology (2010). IT Spiral. http://it-spiral.ist.osaka-u.ac.jp/.

Object Management Group (2006). Ocl 2.0 specification. http://www.omg.org/cgi-bin/apps/doc?formal/06-05-01.pdf.

Owashi, M., Okano, K., and Kusumoto, S. (2008). Design of Warehouse Management Program in JML and Its Verification with Esc/Java2 (in Japanese). *The IEICE Transaction on Information and Systems*, 91(11):2719–2720.

Owashi, M., Okano, K., and Kusumoto, S. (2010). A Translation Method from OCL into JML by Translating the Iterate Feature into Java Methods (in Japanese). volume 27, pages 106–111.

Parr, T. (2007). *The Definitive ANTLR Reference: Building Domain-Specific Language*. Pragmatic Bookshelf.

Rodion, M. and Alessandra, R. (2006). Implementing an OCL to JML translation tool. In *In IEICE Technical Report*, volume 106, pages 13–17.

Sarcar, A. and Cheon, Y. (2010). A new Eclipse-based JML compiler built using AST merging. *Department of Computer Science, The University of Texas at El Paso, Tech. Rep*.