# Bidirectional Translation between OCL and JML for Round-trip Engineering

Hiroaki Shimba ,Kentrao Hanada ,Kozo Okano and Shinji Kusumoto
*Graduate School of Information Science and Technology*
*Osaka University, Japan*
{*h-shimba,k-hanada,okano,kusumoto*} *@ist.osaka-u.ac.jp*

***Abstract*—In recent years, Model-driven development (MDD) based techniques have emerged, and thus translation techniques such as translation from Object Constraint Language (OCL) to Java Modeling Language (JML) have gained much attention. We have been studying not only translation techniques from OCL to JML but also from JML to OCL in order to support Round-trip Engineering (RTE). Two directions of translation among OCL and JML are performed independently without considering unified and iterative translations in our previous work. For an OCL statement and another OCL statement which is obtained from a JML statement which was translated from the original OCL, our previous framework preserves only the meaning of the two statements; however, the forms of the OCL statements may change. It prevents us from RTE-based development. This paper proposes a translation technique between OCL and JML maintaining OCL code by describing their original forms in the comment area of the target languages. Our implementation has been evaluated on two projects used in our previous work and also seven additional open source projects.**

***Keywords*-Model-Driven Development,Bidirectional Translation,OCL,JML,**

## I. Introduction

Agile software development has come to be used recently and has been important in industry. Round-trip Engineering (RTE) is one of agile software development. In RTE, developers generate code from models using MDD techniques and modify the models or the generated code. In general changes in the models or the code are reflected to the corresponding code or the models. Such a developer refines the models and the code using a framework which supports RTE. This framework maintains consistency between the models and the code automatically translating the models to the code and the code to the models. Several approaches to support RTE have been proposed. For example, Tanaka et al. proposed a bidirectional translation technique between Service Oriented Architecture and Process Oriented Architecture [1]. However any bidirectional translation techniques which focus on annotation language such as OCL [2] and JML [3] have not been proposed.

We have been studying not only translation techniques from OCL to JML but also from JML to OCL [4]. The advantage of using the annotation languages such as OCL and JML is described as follows.

- It is possible to describe specification precisely than a natural language.
- It enables a user to detect the location of software defects by the concept of design by contract.
- For JML, there are several tools to check whether inconsistency exists between implemented code and the specification in JML.

In our previous work, two directions of translation among OCL and JML are performed independently without considering unified and iterative translations. For an OCL statement and another OCL statement which is obtained from a JML statement which was translated from the original OCL, our previous framework preserves only the meaning of the two statements, however the forms of the OCL statements may change. This paper proposes a translation technique between OCL and JML with maintaining OCL code by describing their original forms in the comment area of the target languages.

Our contributions are summarized as follows.

- A bidirectional translation technique helps unification of design and implementation by some devices such as describing OCL statements in the comment area of JML such that forms of code are maintained.
- Our implemented tool enables a user to maintain consistency of specification between design and implementation.
- The experimental results highlight the benefits of our translation approach between OCL and JML.

The organization of the remainder of this paper is as follows. Section 2 describes the background of this research. Sections 3, 4, and 5 describe our approach, the implementation of our tool and the experimental results respectively. Finally, Section 6 concludes the paper.

## II. Background

### A. Design by Contract

Design by Contract [5] is known as contract programming. It is one of the concepts of object-oriented designing. It regards specification as a contract between a supplier of the method (callee) and client of the method (caller) for the purpose of improving software quality and reliability.

### B. OCL and JML

OCL is an annotation language for the Unified Modeling Language (UML). It can describe specification more precisely than natural languages. The specification is represented by describing pre-condition, post-condition and invariant condition to model elements of UML. OCL does not change state of models because OCL statements have no side effect.

JML is an annotation language for Java code. It can describe specification of Java methods or objects in the same way as OCL. It is easy to describe the specification in JML because its syntax is similar to that of Java. JML statements are described in the comment area of Java. There are tools to verify Java code annotated with JML. JML Run Time Assertion Checker [6] dynamically detects inconsistency between runtime values and specification in JML. ESCJava2 [7] checks correctness of the implementation of Java code for specification with JML.

### *C. Model Translation*

Model translation in general translates a model in accordance with a certain metamodel into another model in accordance with yet another metamodel. Model translation falls into two types. One translates from a model to another model (M2M), while the other translates from a model to code (M2T). QVT and ATL are typical M2M model translation languages. UML2Java [8] supports M2T translation.

### *D. Round-trip Engineering*

RTE is a software development method performed by repeating forward engineering and reverse engineering. In forward engineering, program code is generated from models such as class diagram and sequence diagram. In reverse engineering, program code is translated into corresponding models. Developers iterate modeling phase and coding phase in RTE. In general, when code or models are modified, changes must be automatically reflected to the corresponding models or the code by using tools which support RTE.

### *E. Xtext*

Xtext [9] is a framework which supports defining syntax of models and translation rules based on M2T. It automatically generates an editor from models. This editor supports code completion and detection of syntax error. If models are described on the editor, they are automatically translated into a target language by using user-defined translation rules.

### *F. Related Work*

An existing method [10] does not adequately support the iterator feature, which is the most basic operation among collection loop operations. Our research group proposed a technique to translate the iterator feature by generating a Java method that is semantically equal to each OCL loop feature [11].

There is a research which generates OCL constraints from a natural language specification[12]. In this research, the natural language which conforms SBVR is translated into OCL. It is useful to study OCL.

Cheon et al. [13] manually translated OCL into four languages such as Java code, assertion, JML and AspectJ. They compared results of verification using translated specification. As a result, they found that Java code and assertiosn are useful from the viewpoint of CPU usage and memory usage. JML and AspectJ are useful in terms of ease of translation and debug.

A translation technique from UML class diagram annotated with OCL to Alloy was proposed by Anastasakis et al [14]. It enables designers of software to verify the UML class diagram. Moreover they proposed a translation technique from the results of verification by Alloy to UML object diagram [15]. Designers can understand the results of the verification without knowledge of Alloy by combining these translations.

It takes a significant amount of time and resources to perform model translation on large and complex models. Razavi et al. [16] proposed an approach for deriving incremental model transformations by partial evaluation of the original model transformation programs. The experimental results indicate that their approach significantly improves the performance of repetitive applications of model transformations.

It is important and difficult to maintain traceability between models and code in MDD. It will be lost when a user of MDD modifies the models or the generated template code. Yu et al. [17] proposed their framework which maintain traceability between the modified models and the modified template code.

## III. Approach

### *A. Translation from OCL to JML*

First, we describe the overview of our proposed translation from OCL to JML. The translation on basic operations is defined in our previous work. The translation rules define one-to-one relation in target elements. The translation on collection loop is performed using the translation rules described at related work [11]. In the bidirectional translation, it is desirable that statements which are not modified by a user are maintained in the same form. Our tool maintains OCL code by inserting their original forms in the comment area in the front of the generated JML statements. Moreover, a hash code for each of the generated JML statements is inserted in the same comment area. It is used to check whether the generated JML statements are modified by a user.

Collection loop operations are replaced with an iterate operation and translated into semantically equivalent Java methods in our previous work. However there are some cases that the translation fail. For example, translation on the statement (1) fails. The keyword "result" means returned value of a method in OCL. In this case, the type of the "result" value is collection. In our previous work, collection

```
/*@ensures translatedIteration1(\result)
                            >= 100; @*/
private Collection sample(){}
private Integer translatedIteration1
         (Collection param_Collection){
    Integer res = 0;
    for (String e : param_Collection){
        res = (res + e);
    }
    return res;
```

Figure 1. translation on iterate operation

which is the target of the iterate operation is translated in the generated methods.

$$result{-}{>}\ iterate(e : Integer; res : Integer = 0 \mid res + e) >= 100 \qquad (1)$$

The keyword "result" in OCL is translated into "\result" in JML. However "\result" is available only in JML statements. The keyword "\result " is not available in Java methods. Thus we perform translation correctly by passing collection (the "\result" value) as argument to the generated methods. The statement (1) is translated successfully into Figure 1.

In our new approach, we maintain OCL statements as original forms in bidirectional translation if generated JML statements are not modified by a user. First, we insert the original OCL statements and its hash code in the comment area in the front of the corresponding JML statements in translation from OCL to JML. Then in reverse translation we calculate the hash code of the target JML statements and compare it with the inserted hash code in order to check whether the generated JML statements are modified by a user. If the target JML statements are not modified, we output the OCL statements in the comment area of the JML statements as a result of translation instead of adapting translation rules on the target JML statements. If the target JML statements are modified, we adapt translation rules. The advantage of this approach is that forms of statements don't become complex by repeated bidirectional translation. Moreover it enables us to translate JML statements which are impossible to translate into OCL unless a user modifies generated code. Figure 2 and 3 describe examples of such translations. Note that the inserted OCL statements as comment in the generated JML statements will be directly outputted in reverse translation.

Basically, OCL statements which are translated from JML statements are translated into JML by using translation rules defined in our previous work. However JML statements which cannot be essentially translated to OCL are maintained original forms inserted in the comment area of OCL statements in translation from JML to OCL. In translation from OCL to JML, these inserted JML statements as comment are directly outputted into JML. Figure 4 describes such situation. The first statement represents JML statement which cannot be essentially translated into an OCL statement. The second statement represents statement which can be translated into JML. The area enclosed by "*" and "*\" is comment in OCL. This approach prevent us from losing information about statements which cannot be essentially translated from JML to OCL.

## B. Translation from JML to OCL

Here, we describe the overview of the translation from JML to OCL firstly. Translation rules on basic operations are defined with one-to-one relation between target elements. However it is necessary to deal with collection loop statements in order to perform translation correctly. Moreover when generated JML statements are not modified by a user, OCL statements which are inserted in the comment area of corresponding JML statements have to be outputted as a result of translation in translation from JML to OCL. There are several JML statements which are impossible to translate into OCL. We need to consider dealing with these statements.

In JML, loop operations such as "exist" and "forall" operation are similar to "for" operations in Java. Thus these operations can handle various variables. However these operations can handle only collection variables in OCL. We assume that the loop operations deal with collection variables in JML. Thus we defined translation rules under this assumption.

In translation from OCL to JML, collection loop operations are translated generating semantically equivalent Java methods. These methods are invoked by statements in JML. We need to analyze the generated methods in order to perform reverse translation.

$$\text{pre} : c{-}{>}\text{iterate(e }:Integer; res : Integer = 0 \mid res + e) >= 10 \qquad (2)$$

$$\text{requires } translatedIteration1() >= 10 \qquad (3)$$

$$\text{pre } : translatedIteration1() >= 10 \qquad (4)$$

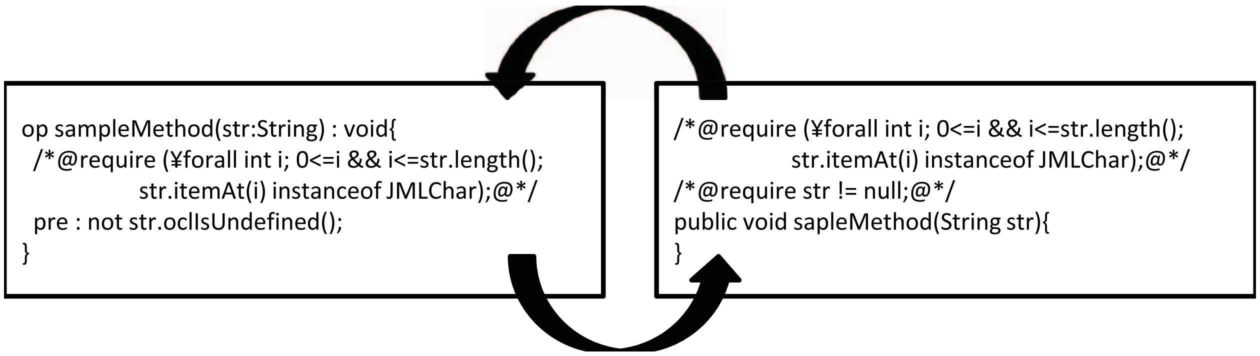
For example, the OCL statement (2) is translated into the JML statement (3) generating a sementically equivalent Java method. The translatedIteration1 method is semantically equal to the statement "*c− >iterate(e : Integer; res : Integer = 0 | res + e)*" . It is invoked by the generated JML statement (3) . Then the statement (3) is translated into the OCL statement (4). As a result, information about implementation of translatedIteration1 in OCL has been lost. Thus we analyze only the generated Java methods semantically equivalent to the original OCL in translation from JML to OCL. Basically modifying these methods are prohibited except for modification in accordance with

```
op getAmount(name : String) : Integer{
  pre  : not name.oclIsUndefined() and name <> ''
  post : itemList->exists(i : Item |
         i.getTotalAmount() = result) or result = 0
}
```

Figure 2. An example of translation between OCL and JML (OCL)

```
//Original_OCL={pre: not name.oclIsUndefined() and name <> ''}
                                        ,JML_HashCode={378505836};
/*@ requires !(name == null) && !(name.equals("")); @*/
//Original_OCL={post: itemlist->exists(i:Item |
                     i.getAmount() = result) or result = 0}
                                        ,JML_HashCode={2016817273};
/*@ ensures translatedIteration3(itemlist, name, \result)
                                                  || \result==0; @*/
public Integer getAmount(String name) {
}
```

Figure 3. An example of translation between OCL and JML (JML)



```
op sampleMethod(str:String) : void{
  /*@require (¥forall int i; 0<=i && i<=str.length();
              str.itemAt(i) instanceof JMLChar);@*/
  pre : not str.oclIsUndefined();
}
```

```
/*@require (¥forall int i; 0<=i && i<=str.length();
            str.itemAt(i) instanceof JMLChar);@*/
/*@require str != null;@*/
public void sapleMethod(String str){
}
```

Figure 4. Translation on inserted JML as comment in OCL

method template. An modifier can change particular part of the methods.

In translation from OCL to JML, the OCL statements and the hash code of the generated JML statements are inserted in the comment area of the JML statements. In translation from JML to OCL we use these information as mentioned in section III-A.

There are JML statements which are impossible to translate into OCL because JML has complex operations than that of OCL. The first statement of Figure 4 is impossible to translate. It is desirable that information on these statements is preserved in bidirectional translation. Thus we insert these JML statements in the comment area of the OCL in translation from JML to OCL, which is reused when translation from OCL to JML is performed.

## IV. Impementation

We implemented a translation tool between OCL and JML using Xtext. First, we define syntax of models such as UML annotated with OCL and Java skeleton code annotated with JML. The syntax of the models is defined by EBNF. Next, we define translation rules on elements of the models. Translations from OCL to JML and JML to OCL are implemented in the same way. The advantage of using Xtext is described as follows.

- Defined syntax of models is reusable for other translation because syntax of the models and translation rules are defined independently.
- An editor which supports code completion and syntax checking is automatically generated from the defined syntax.

## V. Evaluation

### A. Targets and Metrics

We have conducted experiments on nine Java projects annotated with JML statements. Two of the nine projects are developed by our research group in a past research while

Table I
DETAIL OF THE PROJECTS

| Project | Class | JML |
|---|---|---|
| warehouse management | 7 | 142 |
| educational affairs | 200 | 468 |
| PokerTop | 9 | 113 |
| 101JMLSpecifications | 3 | 30 |
| consultorOrtografico | 6 | 124 |
| Zinara | 28 | 126 |
| Lenguajes_III | 16 | 65 |
| P2-master | 5 | 56 |
| extweka | 10 | 233 |
| TOTAL | 77 | 747 |

Table II
TRANSLATION FROM JML TO OCL

| Project | JML | Success | Ratio |
|---|---|---|---|
| Warehouse Management | 142 | 142 | 100% |
| educational affairs | 468 | 444 | 94.5% |
| PokerTop | 113 | 113 | 100% |
| 101JMLSpecifications | 30 | 30 | 100% |
| consultorOrtografico | 124 | 84 | 67.7% |
| Zinara | 128 | 106 | 82.8% |
| Lenguajes_III | 65 | 65 | 100% |
| P2-master | 56 | 56 | 100% |
| extweka | 233 | 230 | 98.7% |
| TOTAL | 1359 | 1270 | 93.5% |

the other seven projects are open source projects obtained from Github. The two projects are a warehouse management system and an educational affairs system. The warehouse management system has correct JML statements as described in the past research. Table I shows the number of the classes and the JML statements each project has.

We measured the following items in the experiments.

- **Ratio of the statements which are successfully translated from JML to OCL (Rsucc)**
  We measured the successfully translated OCL statements from the view of meaning of the statements at translating from JML.
- **Ratio of the semantically corresponding statements in reverse translation (Rseman)**
  We measured the ratio of the generated JML statements which are semantically equal to the original JML statements. Note that forms of the statements may change.
- **Ratio of the syntactically corresponding statements in reverse translation (Rsyntac)**
  We measured the ratio of the generated JML statements which are syntactically equal to the original JML statements. Note that there is no difference except for white space, tab, parentheses and line separators. The ratio of the semantically corresponding statements is greater than the ratio of the syntactically corresponding statements.
- **Ratio of the statements which cannot be translated (Rcannot)**
  We measured the ratio of the statements which cannot be translated from JML to OCL.

Table III
TRANSLATION FROM GENERATED OCL TO JML

| Project | Statements | Semantically | | Syntactically | |
|---|---|---|---|---|---|
| | | Num | Ratio | Num | Ratio |
| Warehouse Management | 142 | 142 | 100% | 62 | 43.7% |
| educational affairs | 444 | 444 | 100% | 214 | 48.2% |
| PokerTop | 113 | 103 | 91.2% | 74 | 65.5% |
| 101JMLSpecifications | 30 | 26 | 86.7% | 9 | 30% |
| consultorOrtografico | 84 | 82 | 97.6% | 63 | 75% |
| Zinara | 106 | 96 | 90.6% | 36 | 34.0% |
| Lenguajes_III | 65 | 65 | 100% | 0 | 0% |
| P2-master | 56 | 30 | 53.6% | 18 | 32.1% |
| extweka | 230 | 200 | 87.0% | 115 | 50% |
| TOTAL | 1270 | 1188 | 93.5% | 591 | 46.5% |

We have conducted the experiments in the following way.

1) We applied the implemented tool to the Java projects annotated with JML statements. It generated UML diagrams annotated with OCL statements.
2) We measured the Rsucc and the Rcannot.
3) We applied the tool to the generated UML dagrams annotated with OCL statements. It generated Java code annotated with JML statements.
4) We measured the Rsyntac and the Rseman.

## B. Results of Experiments

Table II represents the results of the translation from JML to OCL. The Rsucc is 93.5 percent. Most of the remaining statements contain loop operations on an array variables. The array variables in JML are translated into "Sequence" in OCL. Therefore, we define translation rules for statements refering variables with array type. We, however, don't define such rules for statements with loop operations for arrays. In order to deal with such a statement, one possible solution would be to define rules for a subset of such statements, which can be naturally translated into OCL. The solution still has problem on a trade-off between possibility of translation and degrees of freedom of description.

Table III represents the results of the translation from generated OCL to JML. The Rseman is 93.5 percent. There are typos of JML statements in the open source projects. Our tool cannot detect these errors unless the statements conform the syntax of the defined models. If semantical inconsistency between original statements and generated statements occurs, our tool either cannot detect such an error. It is one of the problems we have to deal with in the future. The Rsyntac is 46.5 percent. More than half of the statements change their forms by application of the bidirectional translations which starts from JML. The Rsyntac of Lenguajes_III is zero percent because all statements contain "!=" operator. For example, "*name != null*" becomes "*(name==null)==false*" after application of our bidirectional translation. If a bidirectional translation starts from OCL using our approach, the Rsyntac will be 100 percent unless generated code is modified by a user. If the generated code are modified by a user, our tool translate normally using translation rules defined by our previous work and the statements which cannot be translated "essentially" from JML to OCL are inserted in the comment area of the OCL.

The Rcannot is 6.5 percent. The ratio of the statements which cannot be translated "essentially" in these 6.5 percent statements is 9.86 percent. Thus, the ratio of the statements which cannot be translated "essentially" in all statements is less than one percent. We think it is practical to perform translation between OCL and JML.

## VI. Conclusion

This paper proposes a translation technique between OCL and JML maintaining OCL code by describing their original forms in the comment area of the target languages. We applied our implemented tool to two projects used in our previous work and also seven additional open source projects. From the results of experiment, we found that our tool performs translation from OCL to JML and from JML to OCL efficiently. However syntactically corresponding ratio is not high. We found that the ratio of JML statements which is essentially impossible to translate is less than one percent.

In the future, we want to refine the translation rules in order to improve correspondence ratio because forms of half the number of statements are different from the original statements after applying our several bidirectional translations. We also want to handle other JML features such as pure, model and ghost modifiers.

## Acknowledgments

This work is being conducted as a part of Grant-in-Aid for Scientific Research C(21500036) and S(25220003).

## References

[1] A. Tanaka and O. Takahashi, "Experimental transformations between business process and soa models," *International Workshop on Informatics 2011*, pp. 104–112, 9 2011.

[2] Object Management Group, "Ocl 2.0 specification," 2006, http://www.omg.org/spec/OCL/2.0.

[3] G. Leavens, A. Baker, and C. Ruby, "Jml: A notation for detailed design," *Behavioral Specifications of Businesses and Systems*, pp. 175–188, 1999.

[4] K. Hanada, H. Shinba, K. Okano, and S. Kusumoto, "Implementation of a prototype bi-directinal translation tool between ocl and jml," *International Workshop on Informatics 2012 (IWIN2012)*, pp. 121–127, 9 2012.

[5] B. Meyer, *Eiffel: the language*. Upper Saddle River, NJ: Prentice-Hall, Inc., 1992.

[6] A. Sarcar and Y. Cheon, "A new Eclipse-based JML compiler built using AST merging," *Department of Computer Science, The University of Texas at El Paso, Tech. Rep*, pp. 10–08, 2010.

[7] J. Kiniry and D. Cok, "ESC/Java2: Uniting ESC/Java and JML," *Construction and Analysis of Safe, Secure and Interoperable Smart devices (CASSIS'2004)*, vol. 3362, pp. 108–128, 2005.

[8] W. Harrison, C. Barton, and M. Raghavachari, "Mapping UML designs to Java," in *Proc. of the 15th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, 2000, pp. 178–187.

[9] Eclipse Foundation, "Xtext - Language Development Framework," http://www.eclipse.org/Xtext/.

[10] A. Hamie, "Translating the object constraint language into the java modelling language," in *Proc. of the 2004 ACM symposium on Applied computing*, 2004, pp. 1531–1535.

[11] K. Miyazawa, K. Hanada, K. Okano, and S. Kusumoto, "Class enhancement of our ocl to jml translation tool and its application to a curriculum management system," *In IEICE Technical Report*, vol. 110, no. 458, pp. 115–120, 2011.

[12] M. G. L. I. S. Bajwa, B. Bordbar, "OCL Constraints Generation from Natural Language Specification," in *14th IEEE The Enterprise Computing Conference (EDOC 2010)*, 2010, pp. 204–213.

[13] Y. Cheon, C. Avila, S. Roach, and C. Munoz, "Checking design constraints at run-time using OCL and AspectJ," *International Journal of Software Engineering*, vol. 3, no. 1, pp. 5–28, 2009.

[14] K. Anastasakis, B. Bordbar, G. Georg, and I. Ray, "UML2Alloy: A Challenging Model Transformation," *ACM/IEEE 10th International Conference on Model Driven Engineering Languages and Systems (MoDELS 2007)*, vol. 4735, pp. 436–450, 2007.

[15] S. Shah, K.Anastasakis, and B. Bordbar, "From uml to alloy and back," in *6th Workshop on Model Design, Verification and Validation (MODEVVA 09) published in ACM International Conference Proceeding Series*, vol. 413, 2009, pp. 1–10.

[16] A. Razavi and K. Kontogiannis, "Partial evaluation of model transformations," in *Proc. of the 2012 International Conference on Software Engineering*, ser. ICSE 2012, 2012, pp. 562–572.

[17] Y. Yu, Y. Lin, Z. Hu, S. Hidaka, H. Kato, and L. Montrieux, "Maintaining invariant traceability through bidirectional transformations," in *Proc. of the 2012 International Conference on Software Engineering*, ser. ICSE 2012, 2012, pp. 540–550.