Implementation of a Prototype Bi-directional Translation Tool between OCL and JML

Kentaro Hanada[†], Hiroaki Shinba[†], Kozo Okano[†]and Shinji Kusumoto[†],

[†]Graduate School of Information Science and Technology, Osaka University, Japan
{k-hanada, h-shimba, okano, kusumoto}@ist.osaka-u.ac.jp

Abstract - Object Constraint Language (OCL), which is an annotation language for the Unified Modeling Language (UML), can describe specifications more precisely than can natural languages. In recent years, model-driven architecture (MDA) based techniques have emerged, and thus translation techniques such as translation from OCL to the Java Modeling Language (JML) have gained much attention. Our research group has been studying not only a translation method from OCL to JML but also from JML to OCL. Bi-directional translation between OCL and JML supports (1) development by round-trip engineering (RTE) at the design level, and (2) multi-translations between various formal specification languages. This paper presents our implementations based on model translation techniques.

Keywords: model-driven architecture, OCL, JML, design by contract

1 Introduction

In recent years, model-driven architecture (MDA) [14] based techniques have emerged. MDA targets numerous languages. Thus, translation techniques such as translation from the Unified Modeling Language (UML) to some program languages have gained much attention. Several research efforts have proposed methods that automatically generate Java skeleton files from UML class diagrams [6], [11]. Some of these are publicized as plug-ins for Eclipse. Translation techniques such as the Object Constraint Language (OCL) [20] to the Java Modeling Language (JML) [15] have also been studied. These two languages are described as follows.

- OCL describes detailed properties of UML and is standardized by the Object Management Group (OMG).
- JML specifies properties of a Java program. It is also used in some static program analyzers such as the Extended Static Checker for Java (ESC/Java2) [8].

However, JML describes more detailed properties than does OCL. Both OCL and JML are based on design by contract (DbC) [18] and are able to provide property descriptions of classes or methods.

We previously proposed a method that translates a UML class diagram with OCL into a Java skeleton with JML [19]. Our translation tool is implemented by mapping each of the statements in OCL and JML by a Java program. However, model translation, which uses abstract models to represent common aspects of the target languages, is the primary function of MDA. One of our original goals was providing uniform techniques to translate from OCL to many specification languages. Our previous prototype of a translation tool and other tools provided by other researchers [19], [23] have low reusability, because the goals were fulfillment of translation, not usability. Thus, we need a tool that supports both translation and usability.

This paper presents a prototype translation tool from OCL to JML. First, we define the syntax of UML with OCL by using Xtext, which is a plug-in for Eclipse [5]. Next, we describe the translation rules from UML with OCL to a Java skeleton with JML. The syntax and rules are used for translation in the framework provided by Xtext. The syntax description is independent of the translation rules in Xtext; therefore, the syntax part has high reusability. However, because Xtext can generate a dedicated editor of the defined syntax, this editor has high usability functions, such as code completion and detection of syntax errors.

We also implement a tool that translates from JML to OCL by using the same approach as translation from OCL to JML. Round-trip Engineering (RTE) [17], [25] is a method that gradually refines a model and source code by the repeated use of forward engineering and reverse engineering. The aim of implementation of translation from JML to OCL is to support RTE at the specification description level.

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

2 Background

In this section, we present the background of our research, including techniques and related work.

2.1 Design by Contract

DbC is one of the concepts of object-oriented software designing. The concept regards specifications between a supplier (method) and a client (calling the method) as a contract, with the goal of enhancing software quality, reliability, and reusability. The contract means that if a caller of a class ensures the pre-condition, then the class of the caller must also ensure the post-condition. A pre-condition is a condition that should be satisfied when a method is called. For example, conditions for the arguments of a method are pre-conditions. In contrast, a post-condition is a condition that should be satisfied when a process of a method ends. If the pre-condition is not satisfied, then the caller of its class has errors, and if the post-condition is not satisfied, then the class has errors. These separate responsibilities have a clear distinction for developers, and so they are useful to identify the causes of software defects.

2.2 OCL and JML

OCL, which is standardized by OMG, details the properties of UML models. Because a UML diagram alone cannot express the rich semantics of the relevant information of an application, OCL allows one to describe precisely the additional constraints on the objects and entities present in the UML model.

JML details the constraints of Java methods or objects [15]. These constraints are based on DbC. It is easy for novices to describe the properties in JML because the syntax of JML is similar to that of Java. Various kinds of tools verify source codes with JML annotations. For example, JML Runtime Assertion Checker (JMLrac) [24] checks whether contradictions exist between JML constraints and runtime values of the program. JMLUnit automatically generates a test case skeleton and a test method for JUnit [1]. Since the original use of JML was for runtime assertion checking [4], several other program verification tools have been developed, such as ESC/Java(2) [7], [13], JACK [3], KeY [2], and Krakatoa [16].

2.3 Model Translation

The Query Verification Tool (QVT) [9] and the ATL Transformation Language (ATL) [12] are typical model translation techniques. Model translation has two types. One is Model2Model (M2M) that translates from model to model, and the other is Model2Text (M2T) that translates from model to code. For example, UML2Java [6] provides M2T translation capability.

2.4 Round-trip Engineering

RTE is a method that gradually refines the model and the source code by the repeated use of forward engineering and reverse engineering. RTE development needs to keep the conformity of the models with the source code. By using RTE, QVT feature and requirement changes are easier to make [17], [25]. In general, when the code or models are changed, then the corresponding code or models are changed automatically by using a tool supporting RTE.

2.5 Xtext

Xtext [5] is a support framework for defining both the syntax of a model and the translation rules from the model to the text. Xtext can generate a dedicated editor of the defined syntax. This editor has high usability functions, such as code completion and detection of syntax errors. Moreover, if textual models are written on the editor, the models are automatically translated to text according to the defined translation rules.

2.6 Related Work

Some existing methods [10][23] do not adequately support the iterator feature, which is the most basic operation among collection loop operations. Our research group proposed a technique to resolve this problem by inserting a Java method that is semantically equal to each OCL loop feature [19].

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

Figure 1: General Java template for the iterate feature method

An iterate feature is an operation that applies an expression given as the argument to each element of a collection, which is given as another argument.

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

Expression (1) defines an operation that returns a value representing the sum of all elements in the Set. In expression (1), the first argument ($i : Integer$) defines an iterator variable. The second argument ($sum : Integer = 0$) defines a variable used to store the return value and its initialization. The third argument ($sum+i$) defines the expression executed iteratively in the loop.

In JML or Java, expressions such as "$sum + i$" cannot be evaluated dynamically. For example, if expression (1) is resolved in the same way as expression (2), the result of the translation would be expression (3).

$$\text{JMLTools.flatten}(setOfSets) \quad (2)$$

$$\text{JMLTools.iterate}(\text{int } i, \text{int } sum = 0, sum + i, set) \quad (3)$$

In expression (3), "$sum + i$" is evaluated only once when the method is called. In other words, the expression is not evaluated iteratively and dynamically in every collection element.

To resolve this problem, our research group proposed a technique that inserts a Java method that is semantically equal to each OCL loop feature [22]. It is worthwhile to have such an algorithm to deal with the iterate feature, because the iterate feature is widely used.

Expression (4) shows the general format of an iterate feature. The variables e, $init$, $body$, and c indicate an iterator variable, a declaration of the return value and its initialization, an expression executed in the loop, and a Collection type variable, respectively.

$$c{-}{>} \quad \text{iterate}(e; init \mid body) \quad (4)$$

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

3 Implementation

In this section, we present the implementation of our translation tool.

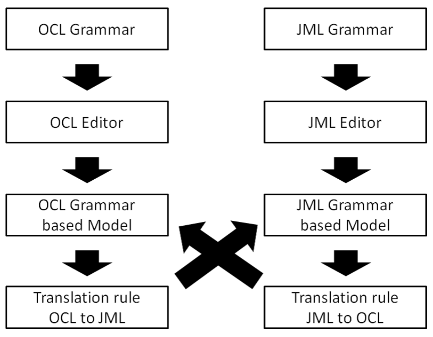
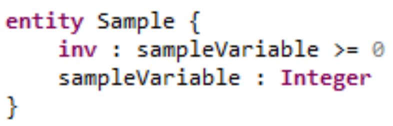
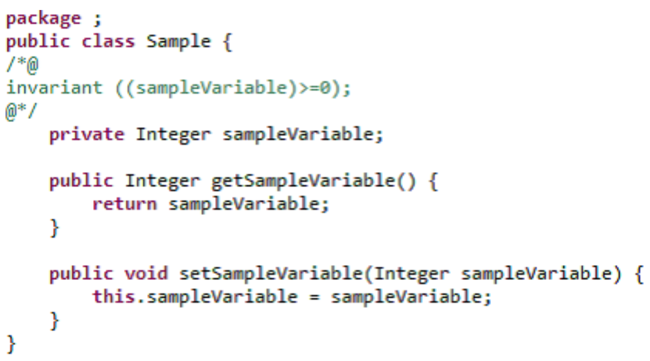


Figure 2: Overview of implementation using Xtext



```
entity Sample {
    inv : sampleVariable >= 0
    sampleVariable : Integer
}
```

Figure 3: Input model



```
package ;
public class Sample {
/*@
invariant ((sampleVariable)>=0);
@*/
    private Integer sampleVariable;

    public Integer getSampleVariable() {
        return sampleVariable;
    }

    public void setSampleVariable(Integer sampleVariable) {
        this.sampleVariable = sampleVariable;
    }
}
```

Figure 4: Result of translation from OCL to JML

3.1 Policy of Implementation

We implement the translation tools by using Xtext. First, we define the syntax of the models. Next, we define the translation rules from the syntax of the models to the source code. Both translations, from OCL to JML and from JML to OCL, are implemented by the above method. Figure 2 shows the overview of the implementation.

Our implementation method has the following advantages.

- Syntax and translation rules are defined independently; thus, the syntax description can be reused.
- Xtext can generate a dedicated editor of the defined syntax. The high usability functions are explained in the previous section.

3.2 Translation from OCL to JML

In this section, we present the implementation of a translation from OCL to JML.

3.2.1 Syntax definition of UML with OCL annotation

We define the syntax of the UML class diagram with OCL. For the UML part, we use conventional syntax rules and extend the syntax. The extended syntax can append the OCL constraints. For the OCL part, we consider some cases of return types and other syntax. Translation rules depend on the syntax of the model; therefore, careful case analysis helps the semantic analysis and enhances the reusability of the syntax of a model. The function of the generated editor depends on the defined syntax. Therefore, the more we take into account the case analysis, the more usability the generated editor has. In summary, careful consideration of the case analysis helps both usability and reusability.

3.2.2 Definition of translation rule from OCL to JML

Table 1 shows parts of the translation rules of OCL to JML. A translation function of an OCL statement to a JML statement is expressed by μ. Here, Integer, Real, and any type of Boolean are expressed by a_i. Any type of Collection is expressed by c_i.

We define the translation rules OCL-JML in accordance with many of the same rules used in existing research [19]. In Table 2, many collection loops can be replaced by iterate features. Therefore, our current research replaces the collection loop with the iterate feature. However, this translation method has some challenges. For example, low readability of the generated code is one challenge. To resolve this problem, if the OCL loop feature directly translates the JML loop feature, we do not replace the collection loop with the iterate feature.

Figure 3 is an example of a textual model based on the defined syntax. Figure 4 is an example of the result of a translation from the model to the text.

3.2.3 Oclvoid Type

The OclVoid type is a class having only the constant named Undefined. The constant is returned when an object is cast into an unsupported type or when a method gets a value from the empty collection. The counterpart of this constant in JML is null. It must be noted that in OCL, a logical expression such as “True or Undefined” is evaluated as an undefined expression, not True. To deal with OclVoid correctly, the translation

Table 1: μ translation rules from OCL to JML

$\mu(a_1 = a_2)$	$=$	$\mu(a_1) == \mu(a_2)$
$\mu(a_1 > a_2)$	$=$	$\mu(a_1) > \mu(a_2)$
$\mu(a_1 < a_2)$	$=$	$\mu(a_1) < \mu(a_2)$
$\mu(a_1 >= a_2)$	$=$	$\mu(a_1) >= \mu(a_2)$
$\mu(a_1 <= a_2)$	$=$	$\mu(a_1) <= \mu(a_2)$
$\mu(a_1 <> a_2)$	$=$	$\mu(a_1)! = \mu(a_2)$
$\mu(c_1 = c_2)$	$=$	$\mu(c_1)$.equals($\mu(c_2)$)
$\mu(c_1 > c_2)$	$=$	$\mu(c_1)$.containsAll($\mu(c_2)$)&&!$\mu(c_1)$.equals($\mu(c_2)$)
$\mu(c_1 < c_2)$	$=$	$\mu(c_2)$.containsAll($\mu(c_1)$)&&!$\mu(c_1)$.equals($\mu(c_2)$)
$\mu(c_1 >= c_2)$	$=$	$\mu(c_1)$.containsAll($\mu(c_2)$)
$\mu(c_1 <= c_2)$	$=$	$\mu(c_2)$.containsAll($\mu(c_1)$)
$\mu(c_1 <> c_2)$	$=$	!$\mu(c_1)$.equals($\mu(c_2)$)
$\mu(c_1$−>size())	$=$	$\mu(c_1)$.size()
$\mu(c_1$−>isEmpty())	$=$	$\mu(c_1)$.isEmpty()
$\mu(c_1$−>notEmpty())	$=$	!$\mu(c_1)$.isEmpty()
$\mu(c_1$−>excludes(a_1))	$=$	$\mu(c_1$−>count(a_1) $= 0)$
$\mu(c_1$−>count(a_1))	$=$	$\mu(c_1$−>iterate(e; acc : Integer $= 0 \mid$ if $e = a_1$ then $acc + 1$ else acc endif))

tool needs to treat OclVoid as follows.

$$(a_1 \;==\; null\,?\,false\;:\;throw\,new\,JMLException())$$

3.3 Translation from JML to OCL

In this section, we present the implementation of the translation from JML to OCL.

3.3.1 Syntax definition of Java skeleton code with JML annotation

We define the syntax of Java skeleton with JML. For Java, we define the syntax of class declaration, class modifier, field variable, and method declaration as the targets of translation. The variable type and others are needed to translate correctly, so we define the syntax of the Java skeleton. For JML, our translation tool can translate a part of the formula defined in the JML Reference Manual. JML is a more detailed language than OCL, and JML has complex expressions that cannot be expressed by OCL. For example, JML has an assignment operation and a shift operation, but OCL does not have either of these operations. At the time of syntax definition, we omit the operations and syntax that cannot be translated from JML to OCL. By omitting syntax that does not support translation from JML to OCL, a user can input only the JML expressions supported by the generated editor. For this reason, it becomes much easier to understand the corresponding syntax.

3.3.2 Definition of translation rule from JML to OCL

Table 3 shows some of the translation rules from JML to OCL. Here, the translation function of an JML statement to a OCL statement is expressed by μ'. Any type of all are expressed by a_i. A type of boolean is expressed by b_i.

In terms of elementary operation, the translation of JML to OCL only has to replace the operator of JML with the operator of OCL. However, to translate correctly, a part of the operator needs to interchange an operand. The syntax of JML is similar to that of Java. For example, the "+ operator" is used in various cases, such as "Integer + Integer" and "String + Integer". OCL does not support operation on different types. In contrast, JML supports "+ operator" involving non-numerical types. In terms of loop operation, exists and forall and other terms are defined as operations of the Collection type in OCL. However, sometimes exists and forall and other terms are used as a for loop of Java in JML. Therefore, the loop operation of JML cannot be translated by the loop operation of OCL. If the loop operation is used as a Collection in JML, our tool translates JML to OCL. If the loop operation is not used as a Collection in JML, our tool outputs error messages.

Table 2: A part of the correspondence table of Collection-Iterate

c_1−>exists($a_1 \mid a_2$)	=	c_1−>iterate(a_1; res : Boolean = false $\mid$ res or a_2)
c_1−>forAll($a_1 \mid a_2$)	=	c_1−>iterate(a_1; res : Boolean = true $\mid$ res and a_2)
c_1−>count(a_1)	=	c_1−>iterate(e; acc : Integer = 0 $\mid$ if $e = a_1$ then $acc + 1$ else acc endif)
st_1−>select($a_1 \mid a_2$))	=	st_1−>iterate(a_1; res : Set(T) = Set {} $\mid$ if a_2 then res −>includeing (a_1) else res endif)
st_1−>reject($a_1 \mid a_2$))	=	st_1−>select($a_1 \mid$ not a_2)
c_1−>any($a_1 \mid a_2$)	=	c_1−>select($a_1 \mid a_2$)−> asSequence()−>first()
c_1−>one($a_1 \mid a_2$)	=	c_1−>select($a_1 \mid a_2$)−>size()= 1

3.4 Type Inference

In OCL, "==" evaluates whether two objects are equivalent. However in JML, "==" evaluates whether two objects are equivalent, and "$equals()$" method evaluates whether two reference types are equivalent. To translate correctly, the variable type, and so on, must be correctly distinguished. When translating from JML to OCL, our tool can distinguish the type information correctly. However, when a user writes a textual model, our tool cannot distinguish the type information.

4 Experiments

This section explains our experiments in detail.

4.1 Overview of Experiments

We conducted two experiments. The goal of the first experiment (Experiment 1) was to evaluate the quality of translation from JML, described as the experimental object, to OCL. The goal of the second experiment (Experiment 2) was to evaluate the quality of translation from OCL, generated by our translation tool, to JML. These experiments were conducted to ensure that our tool has possible applications for RTE.

4.2 Measurements

To evaluate the results of translation, we measured the following two items.

Ratio of Transformation

$$Ratio = OCL_{translated}/JML_{all}$$

Table 3: μ' translation rules from JML to OCL

$\mu'(b_1?b_2{:}b_3\ )$	=	if $\mu'(b_1)$ then $\mu'(b_2)$ else $\mu'(b_3)$ endif
$\mu'(b_1{<==>}b_2\ )$	=	$\mu'(b_1)$= $\mu'(b_2)$
$\mu'(b_1{<=!\ =>}b_2\ )$	=	$\mu'(b_1) <> \mu'(b_2)$
$\mu'(b_1{==>}b_2\ )$	=	$\mu'(b_1)$ implies $\mu'(b_2)$
$\mu'(b_1{<==}b_2\ )$	=	$\mu'(b_2)$ implies $\mu'(b_1)$
$\mu'(b_1\&\&b_2\ )$	=	$\mu'(b_1)$ and $\mu'(b_2)$
$\mu'(b_1 \mid\mid b_2\ )$	=	$\mu'(b_1)$ or $\mu'(b_2)$
$\mu'(b_1 \mid b_2\ )$	=	$\mu'(b_1)$ or $\mu'(b_2)$
$\mu'(b_1\ \hat{}\ b_2\ )$	=	$\mu'(b_1$ xor μ'(b)
$\mu'(b_1\&\ b_2\ )$	=	$\mu'(b_1)$ and $\mu'(b_2)$
μ'(\result)	=	result
μ'(\old(a_1))	=	$\mu'(a_1)$@pre
μ'(\not_modified(a_1))	=	$\mu'(a_1)$ = $\mu'(a_1)$@pre
μ'(\fresh(a_1))	=	$\mu'(a_1)$.oclIsNew()

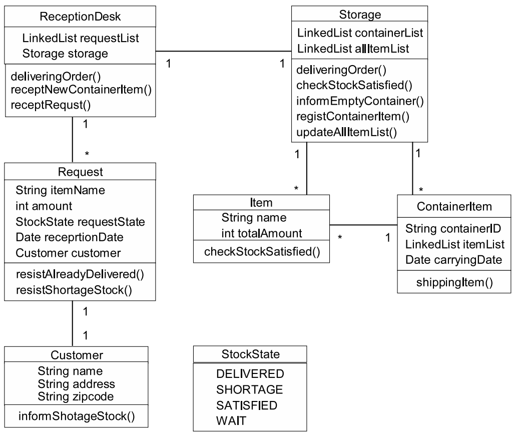


Figure 5: UML class diagram of warehouse management program

Ratio of Reverse Transformation

$Ratio = JML_{reverse}/OCL_{translated}$

JML_{all} is the number of pre-conditions and post-conditions. $OCL_{translated}$ is the number of OCL statements translated from JML statements by our translation tool. $JML_{reverse}$ is the number of JML statements translated from generated OCL statements by our translation tool.

4.3 Results of Experiments

4.3.1 Experiment 1

Experiment 1 uses a warehouse management program. Figure 5 shows the class diagram of the warehouse management program, which consists of seven classes. Table 4 shows the components of the warehouse management program in detail.

The warehouse management program [21] has correct JML statements, as shown by the results of past research [21]. The number of described pre-conditions, post-conditions, and class-invariants is 130. We use these statements to evaluate the quality of the translation. The result shows that the number of correctly translated statements is 102, and the Ratio of Transformation is 78.4%. Figures 6 and 7 show cases of failure translations.

Many cases of failure translations can be found. For example, if multi-variables are declared in the forall feature, then the translation from JML to OCL fails. Additionally, we can classify the following expressions as failures: expressions with type operations, typeof operations, applying ”+” between a String type and numeric type expressions, and so on.

Table 4: 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

```
/*@
ensures \result.matches("containerID." + containerID
          + "CarryingDate | " + carryingDate + "\n{1}")
@*/
String toString(){
}
/*@
ensures (\forall Request r; requestList.contains(r);
         r.getAmount() > 0);
ensures (\forall Request r; requestList.contains(r)
         && r.getAmount() != \old(r.getAmount());
         r.getRequestState() == StockState.SHORTAGE);
@*/
List deliveringOrder(){
}
```

Figure 6: Example of failure translation from JML to OCL (input)

```
context ContainerItem::toString()::String
post : result.matches('ContainerID.'
       [type error][type error][type error][type error])

context ReceptionDesk::deliveringOrder()::List
post : requestList->forAll(r:Request|r.getAmount() > 0)
post : requestList and r=(r)@pre and ->forAll(
       r:getRequestState() = StockState.SHORTAGE)
```

Figure 7: Example of failure translation from JML to OCL (output)

4.3.2 Experiment 2

In Experiment 1, 102 statements are translated correctly. We recheck whether these generated statements are recognized as translation objects of the prototype translation tool from OCL to JML. In terms of correctly translated OCL, the Ratio of Transformation of translation from OCL to JML is 100%. For this reason, translation from JML to OCL by our tool has no problems. However, some bugs are found in the translation from OCL to JML, because our translation rule is still in the trial phase. As a result, 98 statements out of 102 statements as input statements are translated correctly, and the Ratio of Transformation is 96.1%. The result shows that four statements have some bug. Figures 8 and 9 show examples of failure cases.

The OclAsType method is described in the lexical specification. However, the OclAsType method is not described in the translation rules, so our tool could not translate the OclAsType method. After reviewing these results, we modified the method to successfully translate the four statements. Therefore, we will apply our modified translation rule in future work.

5 Discussions

As stated earlier, the result of the Ratio of Transformation of the translation from JML to OCL is 78.4% in Experiment 1. We implemented our tool as a prototype, so our tool has unsupported statements. However, the Ratio of Transformation of the experimental result shows that majority of JML consisted of elementary operations, and thus shows the validity

```
pre : o.oclIsTypeOf(Request)
post : result = (receptionDate.getTime()-
       (o.oclAsType(Request)).getReceptionDate())
       .oclAsType(Integer) or result = 0
op compareTo(o : Object)
```

Figure 8: Example of failure translation from OCL to JML (input)

```
/*@
requires o.getClass().equals(Request);
ensures (\result == (receiptionDate.getTime()-
        ((o.oclAsType(Request)).getReceiptionDate()))
        .oclAsType(Integer)) || (\result == 0);
@*/
public void CompareTo(Object o){
}
```

Figure 9: Example of failure translation from OCL to JML (output)

of our translation tool. We now describe a part of the failure translation.

Our tool could not translate the \type keyword, which is a primitive operator returning a type name. The reason for the above situation is that OCL has no counterpart of the \type operator to identify a type name from a designated expression. To solve this problem, the following approach is considered. First, our tool keeps information on the parameter type before translation from JML to OCL. Next, our tool outputs the parameter type directly in OCL statements.

Result of Ratio of Reverse Transformation is 96.1% in Experiment 2. In Experiment 2, some unsuccessful translated statements also occur in the translation result, because our translation tool from OCL to JML is a prototype. The input OCL is recognized as correct input; therefore, the result shows that the quality of translated OCL is not a problem, but the translation rules have some imperfections.

For this reason, the generated OCL has high quality. Some of the failure translations are due to omissions in the implementation. In terms of this failure translation, our tool will be able to translate correctly with the modified implementation.

Next, we will examine correctness of the rules. Table 1 and 3 show a part of the rules. In general, we have to check that successive application of μ and μ' and vice versa, are preserved. I.e., $\mu'(\mu(o)) = o$ and $\mu(\mu'(j)) = j$ must hold, where (o and j are an OCL expression and a JML expression, respectively). We have manually checked that it holds for every combination of elementary operations. For example, $\mu(\mu'(\backslash result)) = \backslash result$ hold. However, for the iterate operator, some expressions cannot be preserved.

$\mu'(\mu(c_1{-}{>}iterate(a_1; res : \text{Boolean} = \text{false} \mid res \text{ or } a_2)))$
$= \mu'(mPrivateUseForJML01())$
$= mPrivateUseForJML01()$ is a one of such concrete examples. To deal with such expressions is one of our future works.

6 Conclusion

This paper presents a method of implementing the translation from OCL to JML and from JML to OCL. The aim of the implementation of translation from JML to OCL is to support RTE at the specification description level. We applied our tool to a warehouse management program as an experimental object and showed the results of the experiments. One future work is to complete our translation tool, because our tool is at the experimental stage. For example, our tool cannot treat Undefined correctly and needs to be modified.

There are some expressions which cannot be translated correctly by our method including the expressions with iterate operation, and JML loop expressions. To deal with such expressions is one of our future works.

After we improve the implementation of our tool, we will conduct additional experiments. We will again evaluate the quality of translation from OCL to JML and from JML to OCL. We have not yet evaluated the translation from OCL to JML, except for the number of successful translations.

In the future, we will also compare the result of applying generated JML with the review tool for JML and the result of applying described JML manually with the review tool for JML. Two examples of review tools for JML are esc/java2 and jml4c. In terms of the translation tool from JML to OCL, we will compare the generated OCL and the OCL described manually to evaluate the readability. Also, we will apply the generated OCL to the review tool for OCL. One example of a review tool for OCL is Octopus. In addition, we will evaluate whether our tool can do mutual transformations repeatedly by using our translation tool from OCL to JML and from JML to OCL.

7 Acknowledgments

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

REFERENCES

[1] JUnit. http://www.junit.org/.

[2] W. Ahrendt, T. Baar, B. Beckert, M. G. R. Bubel and, R. Hahnle, W. Menzel, W. Mostowski, A. Roth, S. Schlager, and P. Schmitt. The KeY tool. *Software and System Modeling*, 4(1):32–54, 2005.

[3] L. Burdy, A. Requet, and J.Lanet. Java applet correctness: A developer-oriented approach. *K. Araki, S. Gnesi, and D. Mandrioli, editors, FME 2003*, 2805:422–439, 2003.

[4] Y. Cheon and T. Leavens. A runtime assertion checker for the Java Modeling Language (JML). *In Hamid R. Arabnia and Youngsong Mun, editors, the International Conference on Software Engineering Research and Practice (SERP'02)*, pages 322–328, 2002.

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

[6] G. Engels, R.H.ücking, S. Sauer, and A. Wagner. UML collaboration diagrams and their transformation to Java. In *UML1999 -Beyond the Standard, Second International Conference*, pages 473–488, 1999.

[7] C. Flanagan, K. Rustan, M. Leino, M. Lillibridge, G. elson, J. Saxe, and R. Stata. A runtime assertion checker for the Java Modeling Language (JML). *Extended static checking for Java. In ACM SIGPLAN 2002 Conference*

on Programming Language Design and Implementation (PLDI'2002), pages 234–245, 2002.

[8] C. Flanagan, K. Rustan, M. Leino, M. Lillibridge, G. Nelson, J. Saxe, and R. Stata. Extended static checking for Java. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, pages 234–245, 2002.

[9] O. M. Group. Documents associated with meta object facility (mof) 2.0 query/view/transformation, v1.1, 2011. http://www.omg.org/spec/QVT/1.1/PDF/.

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

[11] 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*, pages 178–187, 2000.

[12] F. Jouault, F. Allilaire, J. Bézivin, and I. Kurtev. ATL: A model transformation tool. *Science of Computer Programming*, 72(1-2):31–39, 2008.

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

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

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

[16] C. Marche, C. Paulin-Mohring, and X. Urbain. The KRAKATOA tool for certification of Java/JavaCard programs annotated in JML. *J. Log. Algebr. Program*, 58(1-2):89–106, 2004.

[17] N. Medvidovic, A. Egyed, and D. S. Rosenblum. Round-trip software engineering using uml: From architecture to design and back, 1999.

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

[19] 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*, 110(458):115–120, 2011.

[20] Object Management Group. OCL 2.0 Specification, 2006. http://www.omg.org/cgi-bin/apps/doc?formal/06-05-01.pdf.

[21] M. Owashi, K. Okano, and S. Kusumoto. 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, 2008-11-01.

[22] M. Owashi, K. Okano, and S. Kusumoto. A Translation Method from OCL into JML by Translating the Iterate Feature into Java Methods (in Japanese). *Computer Software*, 27(2):106–111, 2010.

[23] M. Rodion and R. Alessandra. Implementing an OCL to JML translation tool. 106(426):13–17, 2006.

[24] 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*, pages 10–08, 2010.

[25] S. Sendall and J. Küster. Taming model round-trip engineering. In *In Proceedings of Workshop Best Practices for Model-Driven Software Development*, pages 1–13, 2004.

(Received October 15, 2012)
(Revised December 3, 2012)

Kentaro Hanada received the BI degree from Osaka University in 2011. He is a master course student in Osaka University. His research interests include model translation, especially translation between OCL and JML.

Hiroaki Shinba received the BI degree from Osaka University in 2012. He is a master course student in Osaka University. His research interests include model translation, especially translation between OCL and JML.

Kozo Okano received the BE, ME, and Ph.D degrees in Information and Computer Sciences from Osaka University, in 1990, 1992, and 1995, respectively. Since 2002, he has been an associate professor in the Graduate School of Information Science and Technology, Osaka University. In 2002, he was a visiting researcher of the Department of Computer Science, University of Kent at Canterbury. In 2003, he was a visiting lecturer at the School of Computer Science, University of Birmingham. His current research interests include formal methods for software and information system design. He is a member of IEEE_CS, IEICE of Japan and IPS of Japan.

Shinji Kusumoto received the BE, ME, and DE degrees in information and computer sciences from Osaka University in 1988, 1990, and 1993, respectively. He is currently a professor in the Graduate School of Information Science and Technology at Osaka University. His research interests include software metrics and software quality assurance technique. He is a member of the IEEE, the IEEE Computer Society, IPSJ, IEICE, and JFPUG.