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

Kentaro Hanada†, Hiroaki Shimba†, 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 - OCL (Object Constraint Language) is an annotation language for UML. It can describe specification more precisely than natural languages. In recent years, MDA (Model Driven Architecture) based techniques have emerged, thus translation techniques such as translation from OCL to JML (Java Modeling Language) have gained much attention. Our research group has been studying not only a translation method from OCL to JML but also a translation method from JML to OCL. Bi-directional translation between OCL and JML supports (1) development by RTE (Round Trip Engineering) at the design level, and (2) multi-translations among 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, MDA (Model Driven Architecture) [14] based techniques have emerged. MDA targets a lot of languages; thus, translation techniques such as translation from UML (Unified Modeling Language) to some program languages, have gained much attention. Several research efforts have proposed methods which automatically generate Java skeleton files from UML class diagrams [6], [11]. Some of them are publicized as plug-ins for Eclipse. Translation techniques such as OCL (Object Constraint Language) [20] to JML (Java Modeling Language) [15] have been also studied.

- OCL is a language to describe detailed properties of UML and standardized by OMG (Object Management Group).
- JML is a language to specify properties of a Java program. It is also used in some static program analyzers such as ESC/Java2[8].

JML aims for describing more detail properties than OCL does. Both OCL and JML are based on DbC (Design by Contract) [18] and able to provide property descriptions of classes or methods.

We have already proposed a concrete method which translates a UML class diagram with OCL into a Java skeleton with JML [19]. Our translation tool is implemented by mapping each of statements in OCL and JML by Java program. However, model translation which uses some abstract models representing common aspects of the target languages, is the main-stream of MDA. Also one of our original goals is providing uniform techniques to translate from OCL to a lots of specification languages. Our previous prototype of translation tool and other tools provided by other researchers [19], [23] have low reusability. Thus, we consider that we have to develop a useful tool which supports the above issues. And, a major aim of existing tool is fulfillment of translation, so existing tool has low usability.

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

We also implemented a tool which translates from JML to OCL by using the same approach of translation from OCL to JML. Round Trip Engineering (RTE) [17], [25] is a method which gradually refines 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 specification description level.

The organization of the remainder of the paper is as follows. Sec.2 describes the background of this research and related work. Sec.3, 4, and 5 describe the implementation of our tool, experimental results, and discussion, 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 Design by Contract

Design by Contract is one of the concepts about Object Oriented software designing. The concept regards specifications between a supplier (method) and a client (calling the method) as contract. It is introduced to aim at enhancing software quality, reliability and reusability. The contract means that if caller of its class the pre-condition then its class must also ensure the post-condition. A pre-condition is the condition that should be satisfied when a method is called. For example, conditions for the arguments of method are pre-conditions. On the other hands, a post-condition is the condition that should be satisfied when a process of method ends. If the pre-condition is not satisfied then caller of its class has errors and if the post-

condition is not satisfied then class has errors. These separate responsibilities have a clear distinction between the role of developers, and it is useful to distinct the causes of software defect.

2.2 OCL and JML

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

JML is a language to detail constraints of Java methods or objects [15]. 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. There are various kinds of tools to verify the source codes with JML annotations. For example, JML Runtime Assertion Checker (JMLrac) [24] checks that there are no contradictions between JML constraints and runtime values of the program. JMLUnit automatically generates a test case skeleton and a test method for JUnit [1]. The original use of JML was for runtime assertion checking [4]. Several program verification tools are, however, provided such as ESC/Java(2) [7], [13], JACK [3], KeY [2], Krakatoa [16], and so on.

2.3 Model Translation

In order to represent an overview of a system to develop, in usual, a model for the system is used in design phase. For example, UML class diagram is one of such models.

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

2.4 Round Trip Engineering

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

2.5 Xtext

Xtext [5] is a framework to support to define syntax of model and to define translation rule from model to text. Xtext can generate a dedicated editor of the defined syntax. The editor has some high usability functions. For example, code completion, detection of syntax errors and so on. Moreover, if Textual models are written on the editor, the models are translated to text according to defined translation rules automatically.

2.6 Related Work

Some existing methods [10][23] do not enough support iterate feature that 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].

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.

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

Expression (1) defines an operation that returns a value which represents a sum of all elements in Set. In expression (1), 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 it's initialization. The third argument ($sum + i$) stands for an expression that is executed iteratively in the loop.

In JML or Java, expressions like "$sum + i$" cannot be evaluated dynamically. For example, if Expression (1) was resolved by the same way of Expression (2), the result of the translation would be like Expression (3).

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

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

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

Our research group proposed a technique to resolve this problem by inserting a Java method that is semantically equal to each OCL loop feature [22]. It is worthful that the algorithm deals with the iterate feature because an iterate feature is widely used.

Expression (4) 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.

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

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.

```
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

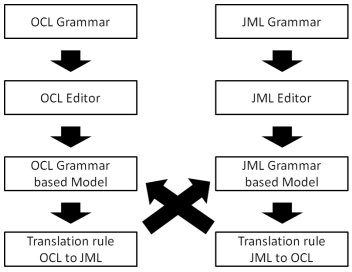

Figure 2: overview of implementation using Xtext

3.1 The policy of Implementation

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

Our implementation method has the following advantage.

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

3.2 Translation from OCL to JML

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

3.2.1 Syntax definition of UML with OCL annotation

We defined syntax of UML class diagram with OCL. In terms of parts of UML, we use a conventional syntax rules, and we extended the syntax. The extended syntax can append OCL constraints. In terms of parts of OCL, we take account of some cases of return type and others. Translation rules depend on syntax of model, therefore careful thought of case analysis helps semantic analysis and enhances utility for reuse syntax of model. The function of the generated editor depend defined syntax. Therefore, the more we take account of case analysis, the more the generated editor has high usability. With all these factors, careful thought of case analysis helps in usability and reusability.

```
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

3.2.2 Definition of translation rule from OCL to JML

Table 1,2 and 3 are a part 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 defined translation rules OCL-JML in much the same rules as the existing research [19]. As Table 4, many collection loops can be replaced by iterate features. Therefore, our existing research replaced the collection loop with the iterate feature. However, this translation method has some challenges. For example, low readability of generated code is one of challenges. In order to resolve this problem, if OCL loop feature directly translates 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 a result of a translation from the model to the text.

3.2.3 Type of Oclvoid

OclVoid type is a class which has only a constant, Undefined. It is returned when an object is casted into an unsupported type or a method gets a value from empty collection. Its counterpart of JML is null. However, in OCL, a logical expression such as "True or Undefined", is not evaluated as undefined expression but True. To deal with OclVoid correctly, the trans-

Table 1: μ translation table of the numeric type

$\mu(a_1 = a_2)$	=	$\mu(a_1) == \mu(a_2)$
$\mu(a_1 > a_2)$	=	$\mu(a_1) > \mu(a_2)$
$\mu(a_1 < a_2)$	=	$\mu(a_1) < \mu(a_2)$
$\mu(a_1 >= a_2)$	=	$\mu(a_1) >= \mu(a_2)$
$\mu(a_1 <= a_2)$	=	$\mu(a_1) <= \mu(a_2)$
$\mu(a_1 <> a_2)$	=	$\mu(a_1)! = \mu(a_2)$

lation tool needs to treat OclVoid as below.

```
(a_1 == null ? false :
        throw new JMLTranslationException())
```

3.3 Translation from JML to OCL

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

3.3.1 Syntax definition of Java skeleton code with JML annotation

We defined the syntax of Java skeleton with JML. In regard to Java, we defined syntax of class declaration, class modifier, field variable and method declaration as target of translation. Variable type and others are needed to translate correctly, so we defined the syntax of Java skeleton. In terms of parts of JML, our translation tool can translate a part of formula that defined in JML Reference Manual. JML is more concrete language than OCL, and JML has complex expression that cannot express by OCL. For example, JML has assignment operation and shift operation, but OCL does not have these operations. At the time of syntax definition, we omitted these operation and syntax that cannot translate from JML to OCL. By omitting syntax that does not support translation from JML to OCL, user can input only JML supported by generated editor. Because of this, it becomes that much easier to understand corresponding syntax.

3.3.2 Definition of translation rule from JML to OCL

Table 5 is a part of translation rules of JML to OCL.

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

Table 2: μ translation table of the collection type

$\mu(c_1 = c_2)$	=	$\mu(c_1)$.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($\mu(c_2)$)

3.4 Type inference

In OCL, "==" is used to evaluate whether or not two objects are equivalent. However in JML, "==" is used in order to evaluate whether or not two reference types are equivalent, and "$equals()$" method is used in order to evaluate whether or not two reference types are equivalent. In order to translate correctly, there is a need to distinguish variable type and so on correctly. When translate from JML to OCL, our tool can distinguish type information correctly. However, when user write textual model, our tool cannot distinguish type information.

4 Experiments

This section will explain experiments in detail.

4.1 Overview of Experiments

We conducted two experiments. The aim of the first experiment (Experiment1) is to evaluate quality of translation from JML that is described in experimental object to OCL. The aim of second experiment (Experiment2) is to evaluate quality of translation from OCL that is generated by our translation tool to JML. It is in order to ensure that our tool has possible application of RTE.

4.2 Measurements

In order to evaluate results of translation, we measured two items of the following.

Ratio of Transformation

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

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 that are translated from JML statements by our translation tool. $JML_{reverse}$ is the number of JML statements that are 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. It consists of seven classes. Table 6 shows components of the warehouse management program in details.

Table 3: μ translation table of the operation of the collection type

$\mu(c_1$->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 | if $e = a_1$ then $acc + 1$ else acc endif))

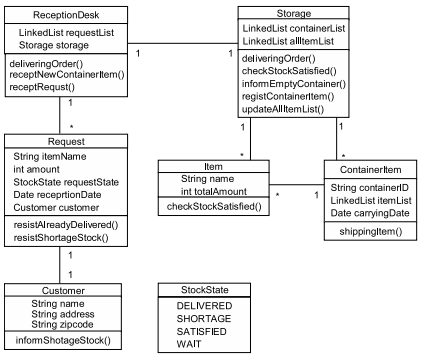

Figure 5: UML class diagram of a warehouse management program

The warehouse management program [21] has correct JML statements by the past research [21]. The number of described pre-condition, post-condition and class-invariant is 130. We used these statements in order to evaluate quality of translation. The result shows that the number of correctly translated statements is 102, Ratio of Transformation is 78.4%. Figure 6, and 7 show the cases of failure translation.

There are many cases of failure translation. For example, if multi-variables are declared in forall feature, then translation from JML to OCL fails. Additionally, we can enumerate the following expressions as other fails : expressions with type operations, typeof operations, applying "+" between String type and numeric type expressions and so on.

4.3.2 Experiment 2

In Experiment 1, 102 statements are translated correctly. We rechecked that these generated statements become recognized as translation object of prototype translation tool from OCL to JML. In terms of correctly translated OCL, Ratio of Transformation of translation from OCL to JML is 100%. For this reason, translation from JML to OCL by our tool has no problem. These are, however, some bugs in translation from OCL to JML, because our translation rule is still in the trial phase.

Table 4: a part of correspondence table of Collection-Iterate

c_1−>exists($a_1 \mid a_2$)	=	c_1−>iterate(a_1; res : Boolean = false | res or a_2)
c_1−>forAll($a_1 \mid a_2$)	=	c_1−>iterate(a_1; res : Boolean = true | res and a_2)
c_1−>count(a_1)	=	c_1−>iterate(e; acc : Integer = 0 | 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 {} | 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

```
/*@
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: An example of a 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: An example of a failure translation from JML to OCL (output)

As a result, 98 statements out of 102 statements as input statements are translated correctly, and Ratio of Transformation is 96.1%. And, the result shows that 4 statements have some bug. Figure 8, and 9 show a part of failure case.

OclAsType method is described in the lexical specification. OclAsType method is not, however, described in the translation rules, so that our tool could not translate oclAsType method. We, however, have the modified method of unsuccessful to translate 4 statements. Therefore we will modify our translation rule in the aftertime.

5 Discussions

Result of Ratio of Transformation is 78.4% in Experiment 1. Here we implemented our tool as prototype, so our tool has unsupported statements. However, Ratio of Transform of experimental result shows that majority of JML are consisted

Table 5: μ translation rule 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()

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

Figure 8: An example of a 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: An example of a failure translation from JML to OCL (output)

of elementary operation. It shows validity of our translation tool. We describe a part of failure translation.

Our tool could not translate \type keyword which is a primitive operator returning a type name. The reason why the above situation happens is that OCL has no counterpart of \type operator to identify a type name from a designated expression. In terms of this problem, the following manner is thought as a solution approach. First, our tool keeps information on 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. There are some unsuccessful translated statements in the result of translation, because our translation tool from OCL to JML is a prototype. Input OCL was recognized as correct input, therefore it shows that a quality of translated OCL has no problem. It shows that translation rules have some imperfection.

For this reason, the generated OCL has high quality. There is some failure translation due to omission of implementation. In terms of this failure translation, our tool will be able to translate correctly by additional implementation.

6 Conclusion

This paper presents concrete method of implementing translation from OCL to JML and reverse translation. The aim of implementation of translation from JML to OCL is to support RTE at specification description level. Also, we applied our tool to a warehouse management program as experimental object and show results of experiments. One of future work is to complete our translation tool. Now, our tool is at the experimental stage, therefore we will implement the rest of our translation tool. For example, our tool cannot treat Undefined correctly and needs to modify on that point. After accomplish implementation of our tool, we will conduct the additional experiment. In terms of evaluation of tool, we will additionally evaluate quality of translation from OCL to JML and from JML to OCL. We have not yet evaluated translation from OCL to JML except for the number of successful translation.

Table 6: 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

For the future, we will make a comparison between result of applying generated JML to review tool for JML and result of applying described JML manually to review tool for JML. As examples of review tool for JML, there is esc/java2, jml4c and so on. In terms of translation tool from JML to OCL, we will make a comparison between generated OCL and described OCL manually to evaluate readability. Also, we will applying generated OCL to review tool for OCL. As examples of review tool for OCL, there is Octopus and so on. Also, we will check carefully to see if our tool can do mutual transformation repeatedly by using our translation tool from OCL to JML and from JML to OCL.

7 Acknowledgments

This work is being conducted as a part of 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.