Round-Trip Engineering Approach to Keep Activity Diagrams Synchronized with Source Code

Keinosuke Matsumoto, Ryo Uenishi, and Naoki Mori
Department of Computer Science and Intelligent Systems
Graduate School of Engineering, Osaka Prefecture University
Sakai, Osaka, Japan
email: {matsu, uenishi, mori}@cs.osakafu-u.ac.jp

Abstract—**Methods for introducing business logic changes and new implementation methods into software should be flexible. Model driven development is regarded as one of the most flexible development methods. The user expects source code to be generated from the models. However, the models and the source code generated from them become unsynchronized if the code is changed. To solve this problem, a round-trip engineering (RTE) approach was proposed that has a feature that keeps the models synchronized with the source code. Tools that provide RTE exist, but almost all of them are applicable only for static diagrams. In this study, RTE is directly adapted to activity diagrams as a type of dynamic diagram. A method is proposed for realizing an RTE approach that keeps activity diagrams synchronized with the source code. The results of application experiments confirmed that the transformation rate of the models and source code is satisfactory. Thus, the success of the proposed RTE approach was verified.**

Keywords-round-trip engineering; model; reverse engineer; activity diagram; model driven development.

I. INTRODUCTION

This paper is based on one [1] presented at ICAS 2015. Model driven architecture (MDA) [2][3] is attracting attention as an approach that can flexibly handle changes in business logics or new software technologies [4][5]. Its core data are models that serve as software design diagrams. It includes features for transforming the models to various types of models and for automatic source code generation [6][7][8][9][10].

The development and standardization of MDA is being advanced by the Object Management Group (OMG). However, the models and the source code generated from them become unsynchronized if the code is changed. To solve this problem, round-trip engineering (RTE) [11][12] [13][14] was proposed. RTE includes a feature that keeps the models synchronized with the source code. Therefore, the consistency between them can be maintained. Tools that provide RTE exist, but almost all of them are applicable only for static diagrams, such as class and component diagrams. Therefore, it is necessary to adapt RTE to dynamic diagrams.

In this study, RTE is adapted to activity diagrams as a type of dynamic diagram. A method is proposed to realize RTE for activity diagrams and source code [15][16]. Activity diagrams are defined in Unified Modeling Language (UML) and express the flows of activities. They can also describe the contents of methods, unlike sequence diagrams. Further, they can express processes hierarchically and are used widely from upper to lower processes of software development. For transforming activity diagrams to source code, the proposed method analyzes the XML metadata interchange (XMI) [17] of the activity entities. XML is a markup language that defines a set of rules for encoding documents in a format that is both human- and machine-readable. XMI is a standard for exchanging metadata information. Conversely, for transforming source code to activity diagrams, the proposed method analyzes the abstract syntax tree (AST) [18] of the source code. In the mutual transformation process, an intermediate representation is used. It has a hierarchical structure and corresponds to both activity diagrams and source code. For this reason, XMI can easily be transformed to an AST and vice versa. For describing conditional branches and loop statements, activity diagrams use the same elements. They cannot be transformed to source code in their existing form. Therefore, a method for analyzing them and transforming one into the other is developed that distinguishes the conditional branches and loop statements. A successful transformation rate of the models and source code was confirmed. Thus, the validity of the proposed method was verified.

The contents of this paper are as follows. In Section II, related work is described. In Section III, the proposed method is explained. In Section IV, the results of application experiments conducted to confirm the validity of the proposed method are given. Finally, in Section V the conclusion and future work are presented.

II. RELATED WORK

This section describes work related to this study.

A. Abstract Syntax Tree

The AST, which belongs to the Eclipse AST implementation, is a directed tree that shows the syntactic analysis results of source code. It is also used to create byte code from the source code as the internal expression of a compiler or interpreter. An AST provides the ASTParser class, which changes source code into an AST. Many types of nodes can be defined by the AST. An AST node can be searched by using the ASTVisitor class corresponding to one of the design patterns [19]. The visitor design pattern allows an algorithm to be separated from the object structure on

which it operates. A practical result of this separation is that new operations can be added to existing object structures without modifying the structures. An example of an AST is shown in Figure 1. A detailed analysis can be performed by changing the AST levels.

B. Consistency among Software Documentations and Source Code

Many approaches, such as those proposed in [20][21], manage consistency and synchronization between software documentation and source code. In particular, RTE refines intermediate results by editing requirement definitions, design plans, and source code alternately. In general, if either the models or code is changed, the RTE automatically reflects the change in the other entity. RTE has a feature that keeps the models synchronized with the source code. An outline of RTE is shown in Figure 2.

Tools, such as UML Lab [22] and Fujaba [23][24], were proposed to maintain the consistency of models and source code. In these tools, a template for generating source code is described by a template description language. Source code can be automatically generated from models by using the template and these tools allow the source code and static diagrams, such as class diagrams and component diagrams, to be refactored synchronously. They also perform code generation and reverse engineering in real time. However, they cannot handle dynamic diagrams, such as activity diagrams, that can describe the behavior of a system. Although Fujaba considers activity diagrams, it does not address them directly. In contrast, our approach can handle activity diagrams directly.

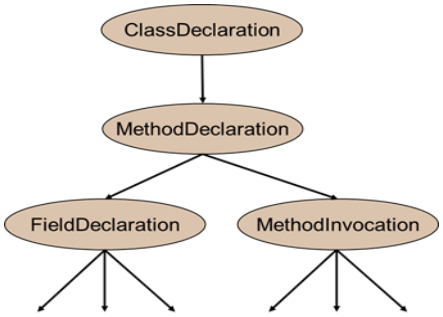


Figure 1. Example of AST.

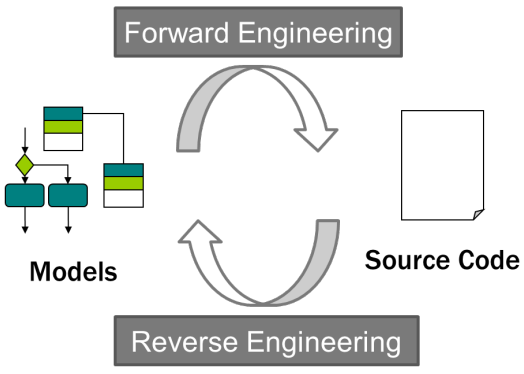


Figure 2. Outline of RTE.

III. Proposed Method

In this section, the proposed transformation method from activity diagrams to source code and from source code to activity diagrams is described. Activity diagrams mainly describe the behaviors of a system using nodes and edges. A content of action is described in a node. The flow of a series of actions is expressed by connecting nodes by edges. An activity diagram is described for each method in class diagrams in the proposed method. Figure 3 shows the basic concept of the proposed method.

A. Transformation from Activity Diagram to Source Code

A specific transformation flow from activity diagrams to source code is as follows.

1) XMI Analysis of Activity Diagram: An activity diagram is expressed in XMI form as a UML file. It begins with a start node and ends with a final node, following nodes or groups through edges. Nodes have information about the actions or controls of the activity diagram. Edges have information about the control flows in the form of attributes and subelements. Group is a tag that has nodes and edges of a subactivity as subelements. Each tag is given an id to discriminate it from other tags. Table I shows the nodes used by an activity diagram.

2) Transformation from XMI to Intermediate Representation: Node and edge tags have a transition starting id and targeting id, respectively. Using these ids, the flow of actions of an activity diagram can be extracted as a sequence of ids. The activity diagram can be transformed from XMI to an intermediate representation by replacing the ids with the corresponding nodes extracted from the XMI analysis. The intermediate representation is a sequence of nodes as the flow of actions. The reason for introducing the intermediate representation is that it facilitates the transformation of XMI to source code and vice versa. Figure 4 shows a metamodel of intermediate representation and Figure 5 shows an example of intermediate representation.

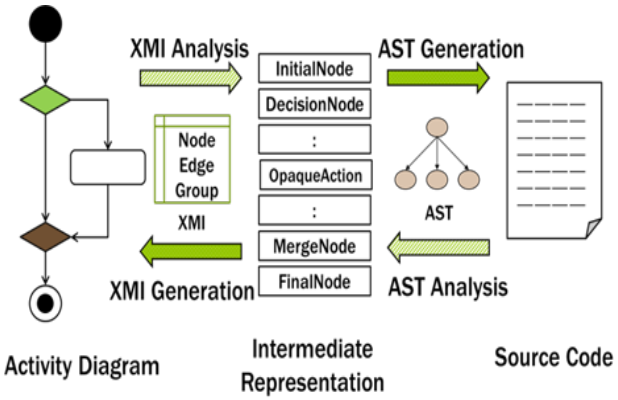


Figure 3. Schematic diagram of the proposed method.

Figure 6 shows an image of this transformation.

3) Transformation from Intermediate Representation to AST: By analyzing the flow of the actions of an intermediate representation, it can be transformed into an AST. The intermediate representation is analyzed in order from the beginning. According to the corresponding nodes, it is necessary to extract information, such as a branch and loop, from the representation structure. For example, a branch has a structure embraced by Decision node and Merge node, but a loop has a structure embraced by Decision nodes. In order to distinguish such structures, a stack that stores the ids of Decision nodes is created. If a Decision node is extracted, the id is pushed to the stack immediately. It is a branch if a Merge node is extracted before a subsequent Decision node is extracted. If a Decision node is extracted and its id is the same id as that popped from the stack, then it is a loop. Otherwise, a new Decision node is extracted and its id is stacked. Figure 7 shows this transformation.

4) Transformation from an AST to Source Code: The target source skeleton code is transformed from class diagrams by using Acceleo templates for classes. Acceleo [25] is the Eclipse Foundation's open-source code generator that provides templates for skeleton code. Transformed activity diagrams and classes of a target source skeleton code are expressed by an AST. A method having a name that is identical to that of an activity diagram can be searched by using ASTVisitor class. The method code transformed from the AST of the activity diagram is added to the method body to which it corresponds in the target source skeleton code for every activity diagram.

TABLE I. NODES USED BY AN ACTIVITY DIAGRAM

Tag	Node
Node tag	ActivityInitialNode
	ActivityFinalNode
	CallBehaviorAction
	CallOperationAction
	DecisionNode
	LoopNode
	MergeNode
	OpaqueAction
Group tag	StructuredActivityNode
Edge tag	ControlFlow

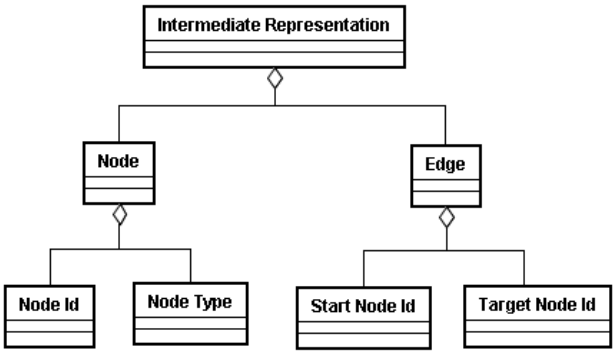


Figure 4. Metamodel of intermediate representation.

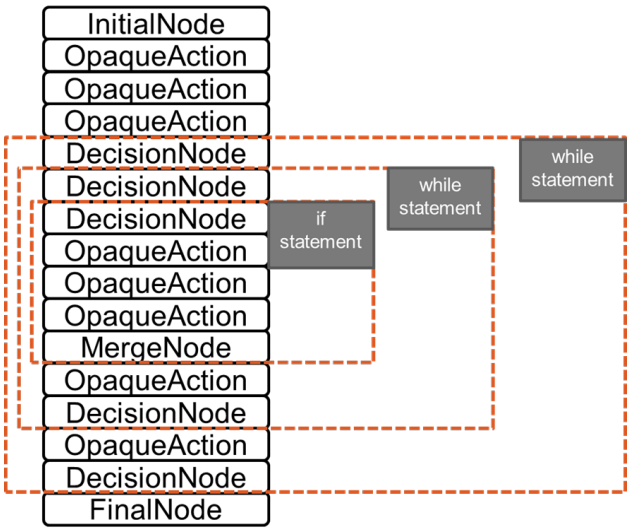


Figure 5. Example of intermediate representation.

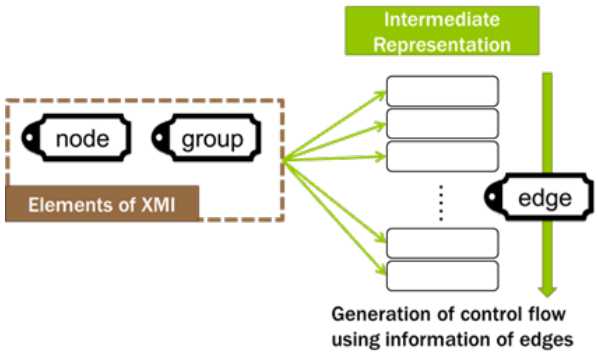


Figure 6. From XMI to intermediate representation.

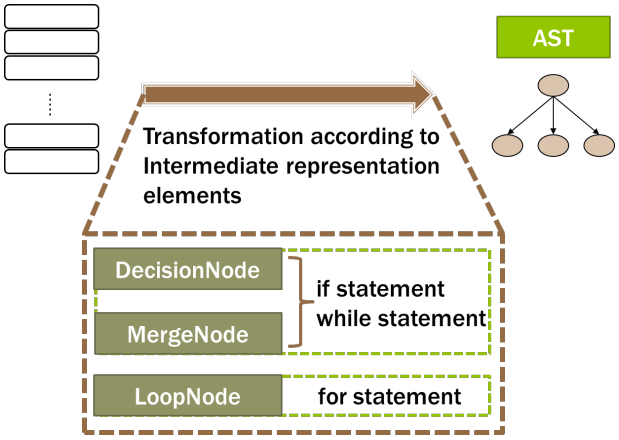


Figure 7. From intermediate representation to AST.

B. Transformation from Source Code to Activity Diagram

The specific flow of transformation from source code to activity diagrams is as follows.

1) AST Analysis of Source Code: The ASTParser class transforms source code into an AST, and the ASTVisitor class searches AST nodes to handle. These are defined as an AST library. The structure of the source code is analyzed by using these classes.

2) Transformation from AST to Intermediate Representation: The required information is extracted by analyzing the AST. Whenever an AST node is searched, the information on the AST node is saved in detail. Required AST nodes are the DeclarationStatement node (such as variables and call of methods) IfStatement node, WhileStatement node, ForStatement node, and so on. The flow of the processing is almost the same as that of the transformation from the XMI of an activity diagram to intermediate representation. Figure 8 shows this transformation.

3) Transformation from Intermediate Representation to XMI: A sequence of ids can be extracted from nodes, groups, and edges in the transformation from activity diagrams to source code. If this transformation is executed in reverse, nodes, groups, and edges are generated by analyzing the flow of actions. Specifically, nodes or groups are generated for each action of the intermediate representation. They are transformed to XML according to the action type. Simultaneously, the edges that connect nodes or groups are

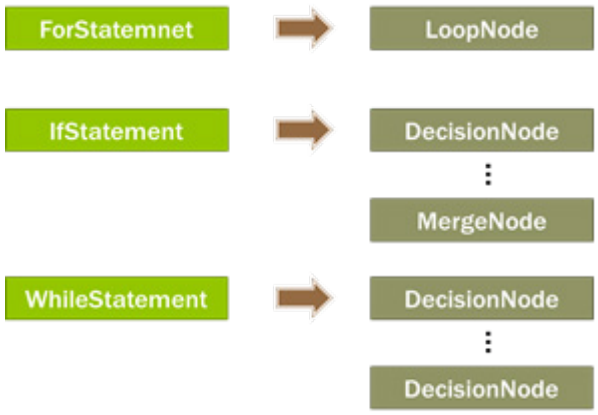


Figure 8. From AST to intermediate representation.

generated. A transition starting id and targeting id can be generated from the sequence of intermediate representation. By generating Decision or Merge nodes expressing branches or loops, a stack that is similar to that of the transformation from activity diagrams to intermediate representation is used.

4) Adding XMI to Activity Diagram: Generated nodes, groups, and edges are added to the XMI file of an activity diagram. In the case of an addition, the user refers to the activity diagram in the package where the source code is located. If the diagram already exists, the addition is performed after deleting the contents of the existing file; otherwise, the addition is performed after generating a new diagram.

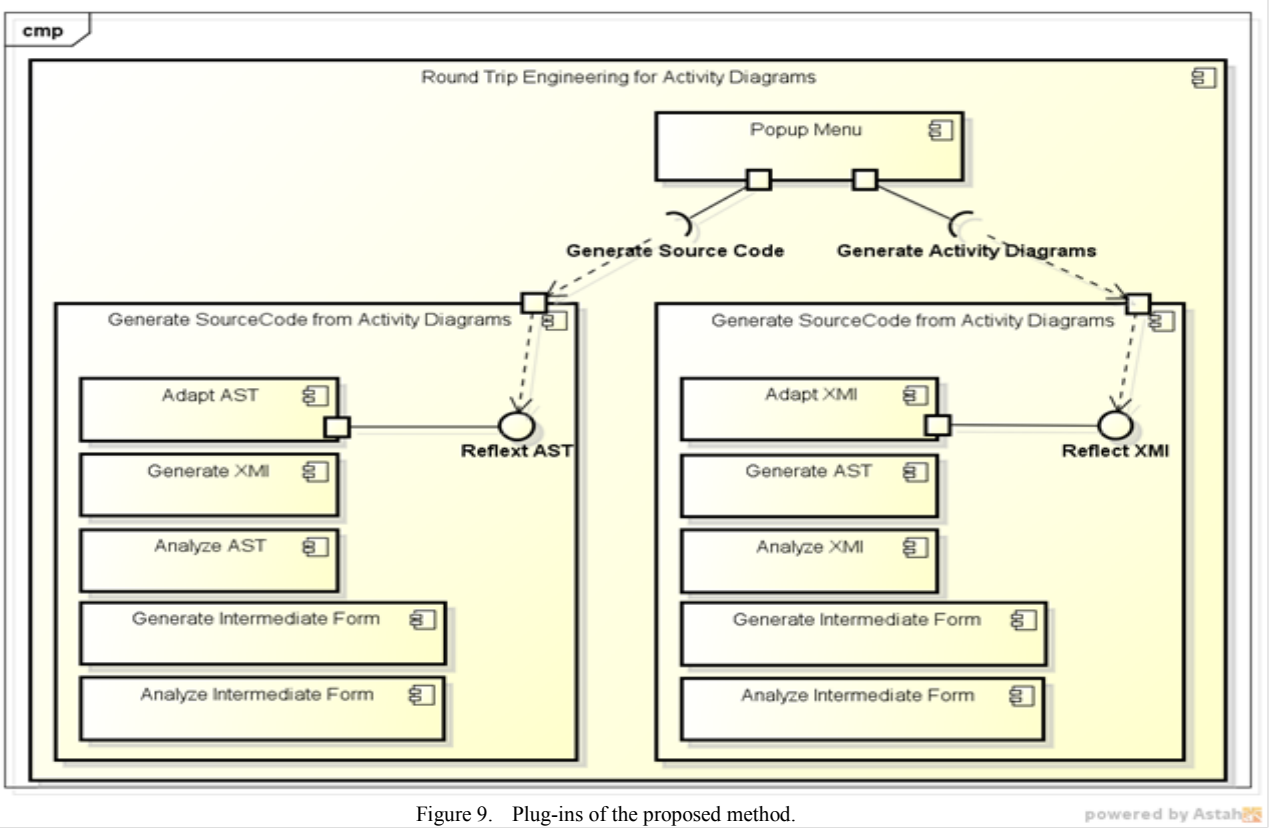


Figure 9. Plug-ins of the proposed method.

C. Implementation of the Proposed Method

The plug-in shown in Figure 9 was implemented in order to realize the proposed method using the integrated development environment, Eclipse [26]. The details of the plug-in are as follows.

1) Reflection of Activity Diagrams in Source Code: The pop-up menu for source code is opened and a dialog reflecting activity diagrams is called. In the dialog, the target activity diagram can be chosen from a selection dialog. An activity that has already been selected can also be made a target in its existing form. Two or more activity diagrams can be chosen from a selection dialog. When an activity diagram is reflected in the source code, the signature of a method is created from the activity diagram. If the signature may agree with that of the method in the target source code, processing is added in the method body.

2) Reflection of Source Code in Activity Diagrams: The pop-up menu for an activity diagram is opened and a dialog reflecting the source code is called. In the dialog, the target source code can be chosen from a selection dialog. A source code that has already been selected can also be made a target in its existing form. When the activity diagram is created, a folder with the name of the target source code is formed in the same package as that which includes the source code. Each activity diagram is created for every method of the source code in the folder. If an activity diagram already exists, overwrite preservation is conducted.

IV. Evaluation of the Proposed Appoach

The proposed method was applied to sample systems to confirm its validity.

A. Transformation Rate

The transformation rate was computed by comparing the number of XMI nodes of the activity diagrams. The objects that were compared were the handwritten activity diagrams and the activity diagrams automatically generated from the source code. The AST and XMI nodes were utilized as an index of whether the transformation had succeeded. The equations for calculating the rate of transformation are as follows.

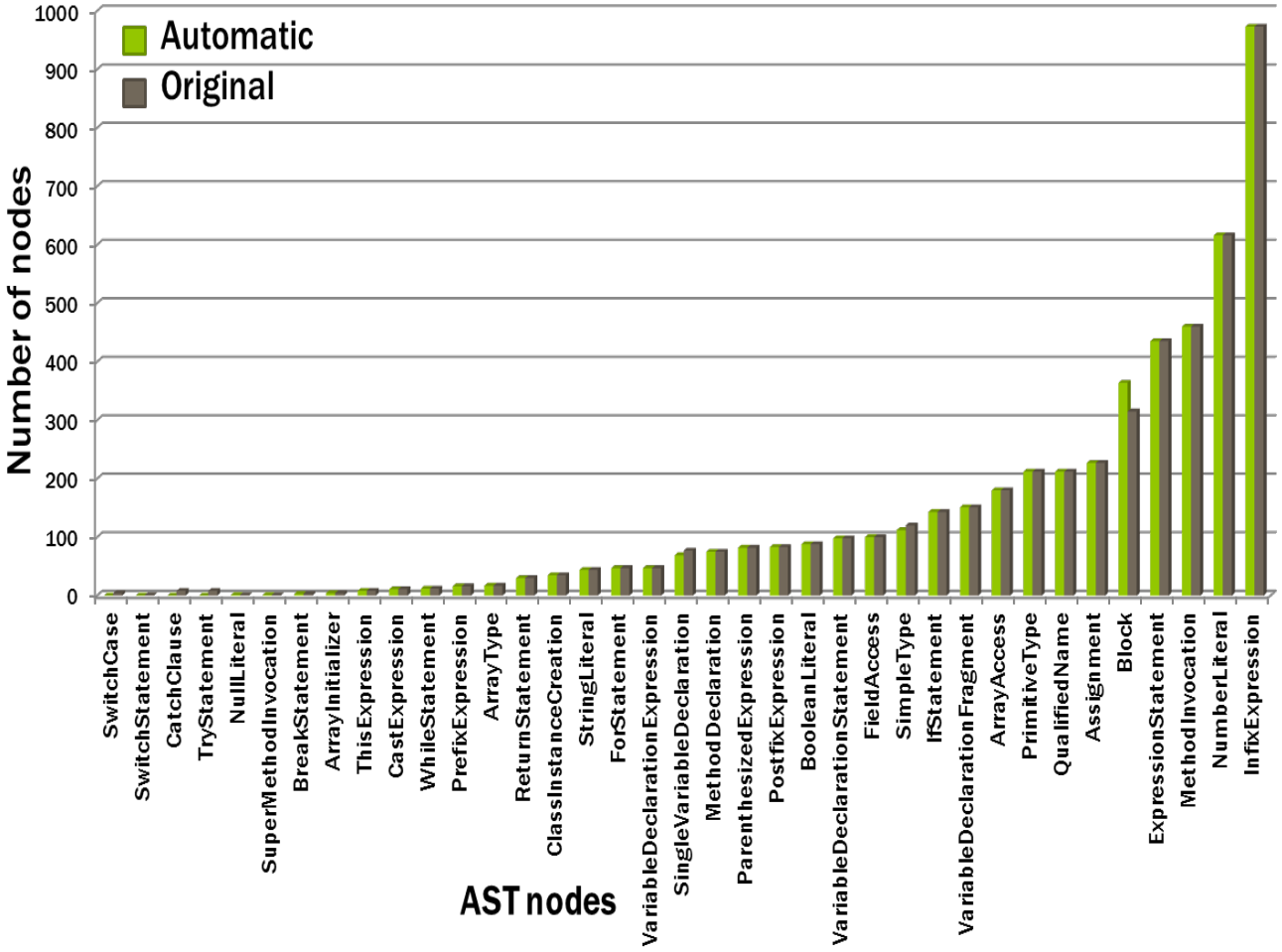


Figure 10. Comparison results of AST nodes.

TABLE II. COMPARISON OF THE NUMBER OF XMI NODES

XMI node	Automatic	Original	Difference
group	47	47	0
packagedElement	75	75	0
guard	155	159	-4
operation	198	198	0
elementImport	300	300	0
importedElement	300	300	0
edge	1137	1142	-5
node	1367	1369	-2

TABLE III. COMPARISON OF THE NUMBER OF AST NODES

AST node	Automatic	Original	Transformation rate (%)
SwitchCase	0	4	0
CatchClause	0	8	0
TryStatement	0	8	0
NullLiteral	1	1	100
SuperMethodInvocation	1	1	100
BreakStatement	3	3	100
ArrayInitializer	4	4	100
ThisExpression	8	8	100
CastExpression	11	11	100
WhileStatement	12	12	100
PrefixExpression	16	16	100
ArrayType	17	17	100
ReturnStatement	30	30	100
ClassInstanceCreation	35	35	100
StringLiteral	44	44	100
ForStatement	47	47	100
VariableDeclarationExpression	47	47	100
SingleVariableDeclaration	69	77	89.6
MethodDeclaration	75	75	100
ParenthesizedExpression	82	82	100
PostfixExpression	83	83	100
BooleanLiteral	88	88	100
VariableDeclarationStatement	98	98	100
FieldAccess	100	100	100
SimpleType	112	120	93.3
IfStatement	147	143	97.2
VariableDeclarationFragment	151	151	100
ArrayAccess	180	180	100
QualifiedName	212	212	100
PrimitiveType	215	212	98.6
Assignment	227	227	100
Block	368	315	83.2
ExpressionStatement	435	435	100
MethodInvocation	460	460	100
NumberLiteral	616	616	100
InfixExpression	977	973	99.6
Total	4971	4944	99.5

$$[1 - \{\text{abs}\,(NA_g - NA_o) / NA_o\}] * 100 \quad (1)$$

where NA_g and NA_o are the number of AST nodes of generated source code and the original source code, respectively.

$$[1 - \{\text{abs}\,(NX_g - NX_o) / NX_o\}] * 100 \quad (2)$$

where NX_g and NX_o are the number of XMI nodes of generated source code and original source code, respectively.

B. Transformation of Hunter Game

The proposed method was applied to a hunter game [27][28]. Both the activity diagrams and source code of the hunter game were available. The number of AST nodes of the original hunter game is 4971. Figure 10 shows the comparison results of the number of AST nodes.

A handwritten activity diagram was transformed into source code and the source code was transformed in reverse into an activity diagram. The two activity diagrams were compared. Tables II and III show a comparison of the number of XMI and AST nodes, respectively.

The XMI nodes that were not transformed can be seen in Table II. There are three types of nodes: guard, edge, and node. A switch statement cannot be described in an activity diagram, but the same processing can be described by using if statements. Guard nodes also decreased in the same number as the switch cases in the generated activity diagrams. The number of edges in connection with them also decreased.

Table III shows the comparison results of AST nodes. The total transformation rate is 99.5%. There are no differences between nodes that in fact express processing, "group," "edge," and "node." This is because only the processing that can be expressed in an activity diagram is expressed in the source code generated from the activity diagram. According to the number of nodes, it was determined that the transformations were successful.

Since there are many kinds of AST nodes, the number of nodes of every type should be compared. The

TABLE IV. COMPARISON OF THE NUMBER OF AST NODES

AST node	Automatic	Original	Difference
ActivityParameterNode	69	69	0
Activity	75	75	0
MergeNode	102	102	0
DecisionNode	114	114	0
InitialNode	118	118	0
ActivityFinalNode	118	118	0
StructuredActivityNode	141	141	0
Expression	159	159	0
CallOperationAction	192	198	-6
PrimitiveType	300	300	0
OpaqueAction	515	509	+6

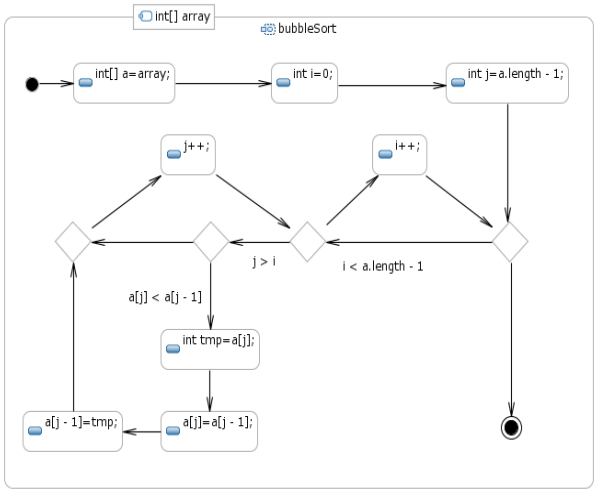
transformation rate is shown in Table IV. Only the number of the node type CallOperationAction decreased. CallOperationAction is an AST node that calls a method. The reason why the number of this node decreased is that the calling method part was not appropriately determined. OpaqueAction expresses the processing that cannot be expressed by the AST nodes. Therefore, OpaqueAction expresses the processing that CallOperationAction should originally express. It can be affirmed that the mutual transformation of activity diagrams was successful.

C. RTE of Activity Diagrams

RTE of activity diagrams and source code was performed using the proposed method. A bubble sort algorithm was used as an example. Figure 11 shows its activity diagram and the source code generated from the diagram. After adding a change (modification, addition, and deletion) to the source code, an activity diagram was generated from the changed source code. It was verified that the generated activity diagram reflected the added change.

1) Code Modification: A modification was added to the generated source code. Items to be modified were variable value, variable name, and loop statement (while statement to for statement). A new activity diagram was generated from the modified source code by the proposed method. Figures 12-13 show the modified source code and activity diagrams for the modifications of variable name and loop statement, respectively. The figures show that the modifications were reflected in the activity diagrams.

2) Code Addition: Code was added to the source code and a new activity diagram was generated from the added source code using the proposed method. Figures 14-15 show the added source code and the activity diagrams for the




```
void bubbleSort(int[] array) {
    int[] a = array;
    int i = 0;
    int j = a.length - 1;
    while (i < a.length - 1) {
        while (j > i) {
            if (a[j] < a[j - 1]) {
                int tmp = a[j];
                a[j] = a[j - 1];
                a[j - 1] = tmp;
            }
            j++;
        }
        i++;
    }
}
```

Figure 11. Original activity diagram and its source code.

```
void bubbleSort(int[] array) {
    int[] a = array;
    int i = 0;
    int j = a.length - 1;
    while (i < a.length - 1) {
        while (j > i) {
            if (a[j] < a[j - 1]) {
                int tmp2 = a[j];
                a[j] = a[j - 1];
                tmp2 = a[j - 1];
            } else {
            }
            j++;
        }
        i++;
    }
}
```

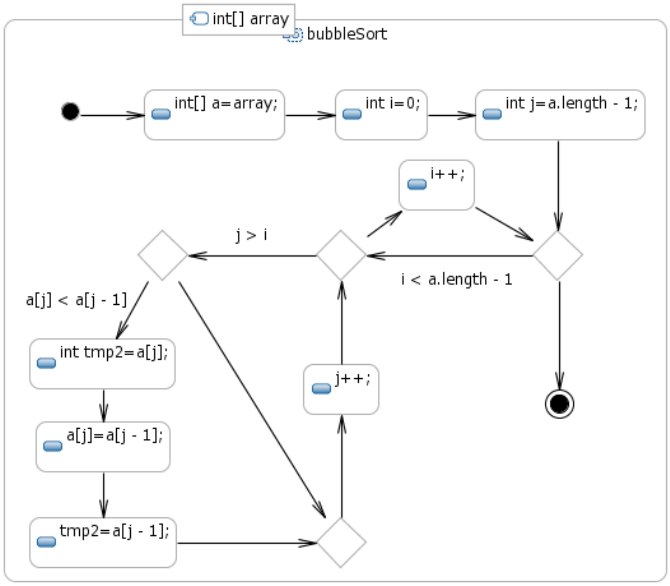


Figure 12. Modified variable code and activity diagram.

addition of a loop statement and if statement. The figures show that the addition was reflected in the activity diagrams as intended.

3) Code Deletion: Code was deleted from the generated source code and a new activity diagram was generated from the deleted source code using the proposed method. Figure 16 shows the deleted source code and the activity diagram for deletion of the if statement. The figure shows that the deletion was reflected in the activity diagram.

```
void bubbleSort(int[] array) {
    int[] a = array;
    int i = 3;
    int j = a.length - 1;

    for (i = 0; i < a.length; i++) {
        for (j = 0; j < i; j++) {
            int tmp = a[j];
            a[j] = a[j - 1];
            tmp = a[j - 1];
        }
    }
}
```

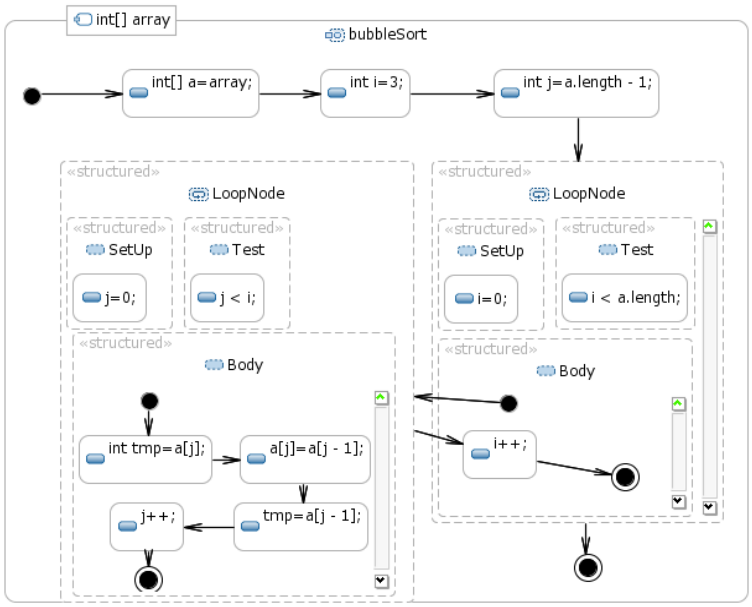


Figure 13. Modified source code and activity diagram.

```
public void bubbleSort(int[] array) {
    int[] a = array;
    int i = 0;
    int j = a.length - 1;
    while (i < a.length - 1) {
        while (j > i) {
            if (a[j] < a[j - 1]) {
                int tmp = a[j];
                a[j] = a[j - 1];
                a[j - 1] = tmp;
            } else {
            }
            j++;
        }
        i++;
    }
    for (int k = 0; k < array.length; k++)
        System.out.println(k);
}
```

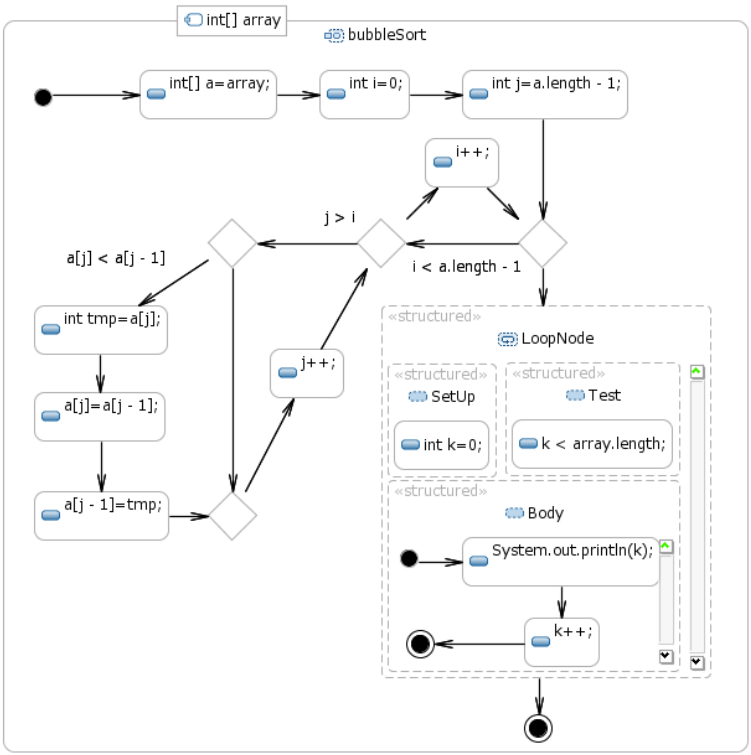


Figure 14. Added source code and activity diagram.

```
void bubbleSort(int[] array) {
    int[] a = array;
    int i = 0;
    int j = a.length - 1;
    while (i < a.length - 1) {
        while (j > i) {
            if (a[j] < a[j - 1]) {
                int tmp = a[j];
                a[j] = a[j - 1];
                a[j - 1] = tmp;
            } else if (a[j] > a[j - 1]) {
                System.out.println(j);
            } else {
            }
            j++;
        }
        i++;
    }
}
```

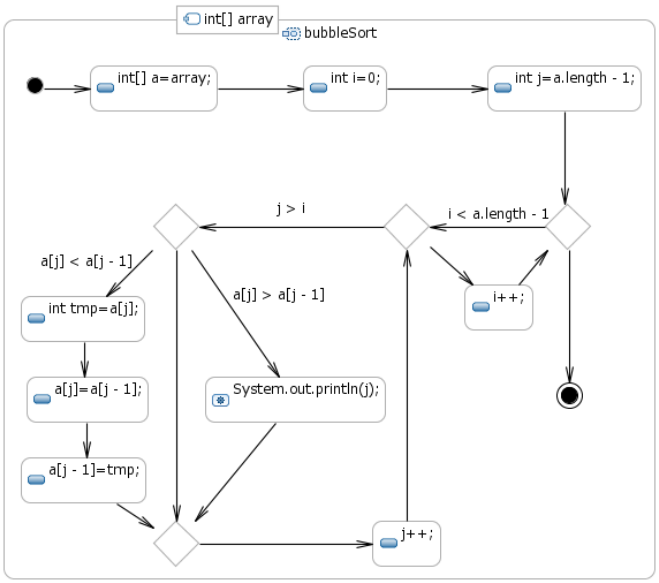


Figure 15. Added source code and activity diagram.

```
void bubbleSort(int[] array) {
    int[] a = array;
    int i = 0;
    int j = a.length - 1;
    while (i < a.length - 1) {
        while (j > i) {
            j++;
        }
        i++;
    }
}
```

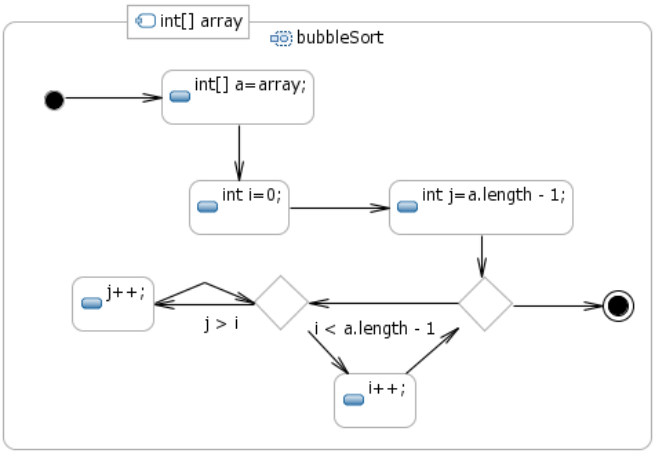


Figure 16. Deleted source code and activity diagram.

Reverse engineering was investigated by generating activity diagrams from handwritten source code and transforming these activity diagrams to source code. The objects that were compared were the handwritten source code and the automatically generated source code. The transformation rate was 99.8%. Except for switch statements and block positions, the two source codes are almost the same. It was verified that the generated source code is functionally equivalent to the handwritten source code. The transformation rates for forward and reverse engineering are not 100%, because there is no standard expression for describing switch and try-catch statements in an activity diagram, and thus, they are not transformed by the proposed method, as shown in Table III.

V. CONCLUSION

In this paper, the problem in model driven development that the models and the source code generated from them become unsynchronized if the code is changed was addressed. In order to solve the problem, an RTE approach was proposed to keep the models used to generated the code updated, i.e., to avoid inconsistences between the systems that are running and giving services and the models used in their development and implementation. Therefore, on the one hand, techniques to translate models into code were proposed, and on the other hand, techniques to obtain models from code were also used. The effectiveness of the proposed method was verified by application experiments using the source code of a hunter game and bubble sort algorithm. It was confirmed that the RTE between the activity diagrams and source code was successful. The characteristics of the

activity diagrams that can be handled by this approach are as follows. They consist of actions of the same granularity, and they do not include many multilayered group nodes.

Since activity diagrams cannot yet handle switch and try-catch statements, the definition of these description methods and increasing the number of convertible elements are important future work.

ACKNOWLEDGMENT

This work was supported in part by JSPS KAKENHI Grant Number 24560501.

REFERENCES

[1] K. Matsumoto, R. Uenishi, and N. Mori, "A round-trip engineering method for activity diagrams and source code," Proc. of the Eleventh International Conference on Autonomic and Autonomous Systems (ICAS 2015), IARIA, May 2015, pp. 24-29, ISBN: 978-1-61208-405-3.

[2] S. J. Mellor, K. Scott, A. Uhl, and D. Wiese, MDA Distilled: Principles of Model Driven Architecture, Addison Wesley Longman Publishing Co., Inc. Redwood City, CA, 2004.

[3] S. Beydeda, M. Book, and V. Gruhn, Model-Driven Software Development, Springer Berlin Heidelberg, 2005.

[4] M. J. Escalona and G. Aragón, "NDT: A model-driven approach for Web requirements," IEEE Transactions on Software Engineering, vol. 34, no. 3, 2008, pp. 377-390.

[5] G. Casale, D. Ardagna, M. Artac, F. Barbier, E. Di Nitto, A. Henry, G. Iuhasz, C. Joubert, J. Merseguer, V. I. Munteanu, J. F. Pérez, D. Petcu, M. Rossi, C. Sheridan, I. Spais, D. Vladušič, "DICE: Quality-driven development of data-intensive cloud applications," Proc. of the Seventh International Workshop on Modeling in Software Engineering (MiSE 2015), May 2015, pp. 1-6.

[6] A. Uhl, "Model-Driven development in the Enterprise," IEEE Software, January/February 2008, pp. 46-49.

[7] R. F. Paige and D. Varró, "Lessons learned from building model-driven development tools," Software System Model, vol. 11, 2012, pp. 527-539.

[8] N. Condori-Fernández, J. I. Panach, A. I. Baars, and T. Vos, Ó. Pastor, "An empirical approach for evaluating the usability of model-driven tools," Science of Computer Programming, vol. 78, no. 11, 2013, pp. 2245–2258.

[9] K. Matsumoto, T. Maruo, M. Murakami and N. Mori, "A graphical development method for multiagent simulators," modeling, simulation and optimization - focus on applications, Shkelzen Cakaj, Eds., March 2010, pp. 147-157, INTECH, ISBN 978-953-307-055-1.

[10] K. Matsumoto, T. Mizuno, and N. Mori, "A method of applying component-based software technologies to model driven development," Proc. of the Third International Conference on Intelligent Systems and Applications (INTELLI 2014) IARIA, June 2014, pp. 54-59, ISBN: 978-1-61208-352-0,

[11] N. Medvidovic, A. Egyed, and D. S. Rosenblum, "Round-trip software engineering using UML: From architecture to design and back," Proc. of the 2nd Workshop on Object Oriented Reengineering, 1999, pp.1-8.

[12] U. Aßmann, "Automatic roundtrip engineering," Electronic Notes in Theoretical Computer Science, vol. 82, 2003, pp. 33-41.

[13] A. Henriksson and H. Larsson, "A definition of round-trip engineering," Technical Report, University of Linköping, Sweden, 2003.

[14] M. Antkiewicz and K. Czarnecki, "Framework-specific modeling languages with round-trip engineering," in Model Driven Engineering Languages and Systems, Springer Berlin Heidelberg, 2006, pp. 692-706.

[15] A. K. Bhattacharjee and R. K. Shyamasundar, "Activity diagrams: A formal framework to model business processes and code generation," Journal of Object Technology, vol. 8, no. 1, January-February 2009, pp. 189-220 .

[16] Pakinam N. Boghdady, Nagwa L. Badr, Mohamed Hashem, and Mohamed F. Tolba, "A proposed test case generation technique based on activity diagrams," International Journal of Engineering & Technology IJET-IJENS vol. 11, no. 3, 2011, pp. 35-52.

[17] XML metadata interchange. XMI: [Online]. Available from: http://www.omg.org/spec/XMI/ 2015.11.24.

[18] I. Neamtiu, J. S. Foster, and M. Hicks., "Understanding source code evolution using abstract syntax tree matching," ACM SIGSOFT Software Engineering Notes, vol. 30, no. 4. ACM, 2005, pp. 1-5.

[19] E. Gamma, R. Helm, R. Johson, and J. Vlissides, Design patterns: Elements of reusable object-oriented software, Addison-Wesley, 1995.

[20] A. Cicchetti, D. D. Ruscio, R. Eramo, and A. Pierantonio, "Automating co-evolution in model-driven engineering," Proc. of the 12th International Enterprise Distributed Object Computing Conference (EDOC'08), IEEE, September 2008, pp. 222-231.

[21] Roberto E. Lopez-Herrejon and Alexander Egyed, "C2mv2: Consistency and composition for managing variability in multi-view systems," Proc. of the 15th European Conference on Software Maintenance and Reengineering. IEEE, 2011. pp. 347-350.

[22] Unified Modeling Language Lab. UML Lab: [Online]. Available from: http://www.uml-lab.com/en/uml-lab/ 2015.11.24.

[23] U. A. Nickel, J. Niere, J. P. Wadsack, and A. Zündorf, "Roundtrip engineering with Fujaba," Proc. of the Second Workshop on Software-Reengineering (WSR), Bad Honnef, 2000, pp. 1-4.

[24] L. Geiger and A. Zundorf, "Tool modeling with Fujaba," Electronic Notes in Theoretical Computer Science, vol. 148, 2006, pp. 173-186.

[25] Acceleo: [Online]. Available from: http://www.eclipse.org/acceleo/ 2015.11.24.

[26] Eclipse: [Online]. Available from: https://www.eclipse.org/home/index.php 2015.11.24.

[27] M. Benda, V. Jagannathan, and R. Dodhiawalla, "On Optimal Cooperation of Knowledge Sources," Technical Report, BCS-G 2010-28, Boeing AI Center, 1985.

[28] K. Matsumoto, T. Ikimi, and N. Mori, "A switching Q-learning approach focusing on partial states," Proc. of the Seventh IFAC Conference on Manufacturing Modelling, Management, and Control, June 2013, pp. 982-986, Saint Petersburg, Russia.