A Round-Trip Engineering Method for Activity Diagrams and 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*—In the field of software development, many implementation methods appear one after another. It is necessary for them to be flexibly introduced into software. Model driven development is regarded as one of the most flexible development methods. It expects to generate source code from the models. However, the models and the source code generated from them will become out of sync if the code is changed. In order to solve this problem, round-trip engineering (RTE) has been proposed. RTE has a feature that keeps the models synchronized with the source code. There are some tools providing us with the RTE, but almost all of them are applicable only for static diagrams. This research adapts the RTE directly to activity diagrams as one of dynamic diagrams, and proposes a method to realize the RTE for activity diagrams and source code. A success transformation rate of the models and source code has been confirmed. As a result, it could be verified that the round-trip engineering between activity diagrams and source code is successful.**

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

I. INTRODUCTION

Model driven architecture (MDA) [1][2] draws attention as a technique that can flexibly deal with changes of business logics or implementation technologies in the field of system development. Its core data are models that serve as design diagrams of software. It includes a transformation to various kinds of models and an automatic source code generation from the models [3][4][5].

Development standardization is advanced as model driven architecture by Object Management Group (OMG). However, the models and the source code generated from them will become out of sync if the code is changed. In order to solve this problem, round-trip engineering (RTE) [6][7][8][9] has been proposed. RTE has a feature that keeps the models synchronized with the source code. Therefore, it is possible to keep them consistent. There are some tools providing the RTE, but almost all of them are applicable only for static diagrams such as class diagrams, component diagrams. Therefore, it is necessary to adapt the RTE to dynamic diagrams.

This research adapts the RTE to activity diagrams as one of the dynamic diagrams, and proposes a method to realize the RTE for activity diagrams and source code [10][11]. Activity diagrams are defined in Unified Modeling Language (UML), and describe flows of activities. They can also express processes hierarchically and are used widely from upper to lower processes of software development. Figure 1 shows a basic concept of the proposed method. In transforming activity diagrams to source code, the proposed method analyzes XML metadata interchange (XMI) [12] of the activity entities. XML is a markup language that defines a set of rules for encoding documents in a format which is both human-readable and machine-readable. XMI is a standard for exchanging metadata information. Conversely, in transforming source code to activity diagrams, the proposed method analyzes the abstract syntax tree (AST) [13] of the source code. In mutual transformation of them, an intermediate representation is used. It has hierarchical structure, and corresponds to both activity diagrams and source code. For this reason, you can easily transform between XMI and AST. Describing conditional branches and loop statements, activity diagrams use the same elements. They cannot be transformed to source code as they are. Therefore, a method for analyzing them and mutual transforming is developed for distinguishing the conditional branches and loop statements. A success transformation rate of the models and source code has been confirmed. As a result, it could be verified that the validity of the proposed method.

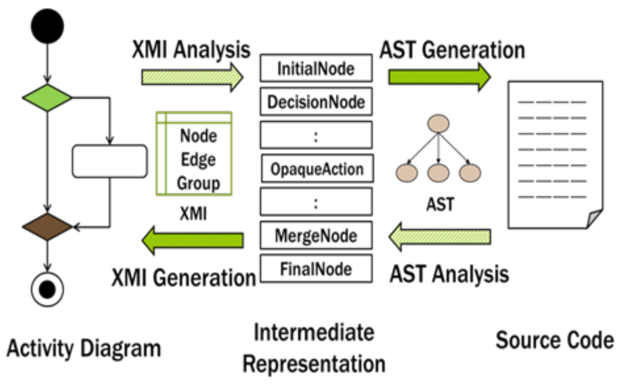


Figure 1. Schematic diagram of the proposed method.

The contents of this paper are shown below: Section II describes related work. Section III explains the proposed method of this research. Section IV shows the results of application experiments in order to confirm the validity of the proposed method. Finally, Section V describes conclusion and future work.

II. Related Work

This study uses related work called AST and RTE.

A. Abstract Syntax Tree

AST that belongs to Eclipse AST implementation is a directed tree showing the syntactic analysis results of source code. It is also used in order to create byte code from the source code as internal expression of a compiler or an interpreter. AST provides us with ASTParser class which changes source code into AST. There are many kinds of nodes defined by AST. An AST node can be searched by using ASTVisitor class corresponding to one of design patterns [14]. The visitor design pattern is a way of separating an algorithm from an object structure on which it operates. A practical result of this separation is the ability to add new operations to existing object structures without modifying those structures. An example of AST is shown in Figure 2. Detailed analysis can be carried out by changing AST levels.

B. Round-Trip Engineering

RTE refines intermediate results by editing requirement definitions, design plans, and source code alternately. Generally, if either models or code is changed, the RTE automatically reflects the change on the other side. RTE has a feature that keeps the models synchronized with the source code. The outline of RTE is shown in Figure 3.

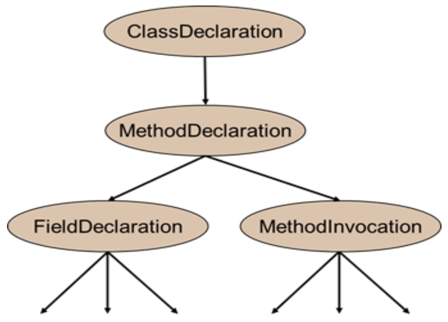


Figure 2. An example of AST.

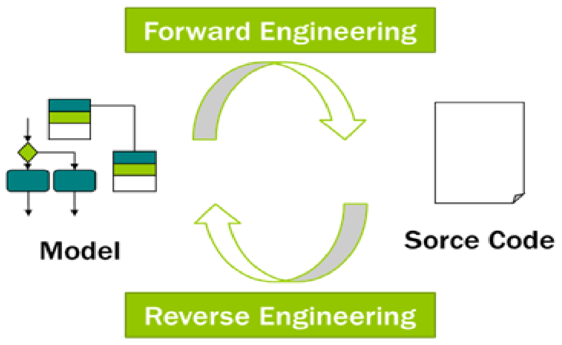


Figure 3. Outline of RTE.

Some tools, like UML Lab [15] and Fujaba [16][17], are proposed to maintain consistency of models and source code. In these tools, a template for generating source code is described by a template description language. Automatic generation of source code can be carried out from models by using the template. It enables to refactor source code and static diagrams, such as class diagrams and component diagrams, synchronously. It also does code generation and reverse engineering in real time. However, it does not deal with dynamic diagrams like activity diagrams which can describe the behavior of a system. Although Fujaba considers activity diagrams, the tool does not address them in a direct way. On the other hand, our approach deals directly with the activity diagrams.

III. Proposed Method

This section proposes a transformation method from activity diagrams to source code and from source code to activity diagrams. 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.

A. Transformation from Activity Diagram to Source Code

A concrete 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 an UML file. It begins with a start node and ends with a final node, following some nodes or groups through edges. Nodes have information on actions or controls of the activity diagram. Edges have information on control flows as some attributes and subelements. Group is a tag that has nodes and edges of a subactivity as subelements. Each tag is given an id for discriminating from other tags. Table I shows 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, you can extract the flow of actions of an activity diagram as a sequence of ids. It can be transformed to an intermediate representation by replacing ids with corresponding nodes

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

extracted from XMI analysis. The intermediate representation is a sequence of nodes as the flow of actions. The reason for introducing the intermediate representation is because it makes it easy to transform both XMI and source code into one another. Figure 4 shows a metamodel of intermediate representation, and Figure 5 shows the image of this transformation.

3) Transformation from Intermediate Representation to AST: Analyzing the flow of the actions of an intermediate

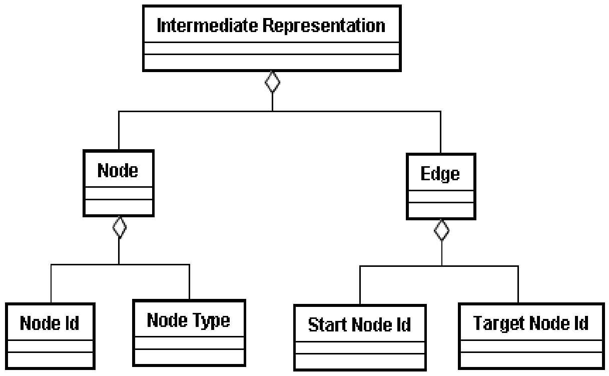


Figure 4. Metamodel of intermediate representaion.

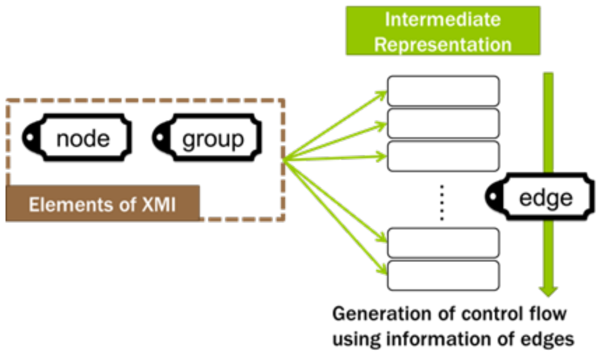

Intermediate Representation
node
group
Elements of XMI
edge
Generation of control flow using information of edges

Figure 5. From XMI to intermediate representation.

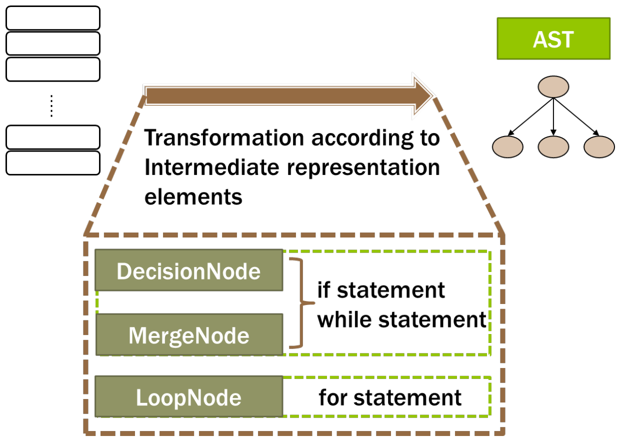


Figure 6. From intermediate representation to AST.

representation, you can transform it into AST. The inter mediate representation is analyzed in order from the beginning. According to 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 which stores ids of Decision nodes is created. If a Decision node comes out, the id is pushed to the stack at once. It is a branch if a Merge node comes out before a Decision node comes out next. If a Decision node comes out and its id is the same id pop from the stack, then it is a loop. Otherwise, a new Decision node comes out and its id is stacked. Figure 6 shows this transformation.

4) Transformation from AST to Source Code: Target source skeleton code is transformed from class diagrams by using Acceleo templates for classes. Acceleo [18] is the Eclisp Foundation's open-source code generator which provides us with templates for skeleton code. Transformed activity diagrams and classes of a target source skeleton code are expressed by AST. A method whose name is identical with that of an activity diagram can be searched by using ASTVisitor class. The method code transformed from AST of the activity diagram is added to the method body to which corresponds in the target source skeleton code for every activity diagram.

B. Transformation from Source Code to Activity Diagram

A concrete flow of transforming from source code to activity diagrams is as follows:

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

2) Transformation from AST to Intermediate Representation: Required information is extracted by analyzing AST. Whenever an AST node is searched, the information on the AST node is saved in detail. Required AST nodes are DeclarationStatement node (like variables, call of methods), IfStatement node, WhileStatement node, ForStatement, and so on. The flow of the processing is almost the same as that of the transformation from XMI of an activity diagram to intermediate representation. Figure 7 shows this transformation.

3) Transformation from Intermediate Representation to XMI: A sequence of ids could be extracted from nodes, groups, and edges in the transformation from activity diagrams to source code. If this transformation is carried on 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 kind of actions. Simultaneously, edges which connect between nodes or groups are generated. A transition starting id and targeting id are generable from the sequence of intermediate representation. Generating Decision or Merge nodes expressing branches or loops, a stack which 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 XMI file of an activity diagram. In case of adding, you refer to the activity diagram in the package where the source code is allocated. If the diagram already exists, adding is performed after deleting the contents of the existing file. Otherwise, adding is performed after generating a new diagram.

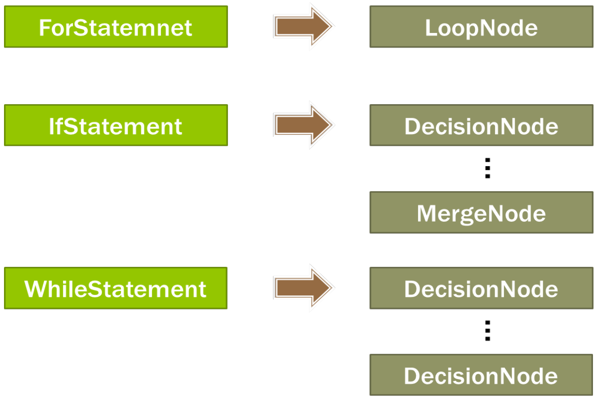


Figure 7. From AST to intermediate representation.

IV. Application Experiments

The proposed method is applied to a hunter game [19] to confirm the effectiveness of the proposed method. We have both activity diagrams and source code of the hunter game. The number of AST nodes of original hunter game is 4971. Mutual transformations of the activity diagrams and the source code are carried out by the proposed method. As a result, Figure 8 describes comparison results of the number of AST nodes. Tables II and III show the comparison of the number of XMI and AST nodes respectively.

The transformation rate is computed by comparing the number of XMI nodes of activity diagrams. The objects to compare are handwritten activity diagrams and the activity diagrams automatically generated from the source code.

TABLE II. Comparison of the number of XMI nodes.

XMI node	Automatic	Original	Difference
group	47	47	0
guard	155	159	-4
edge	1137	1142	-5
node	1367	1369	-2

TABLE III. Comparison of the number of AST nodes.

AST node	Automatic	Original
SwitchCase	0	4
SwitchStatement	0	1
CatchClause	0	8
TryStatement	0	8
VariableDeclaration	69	77
Block	364	315

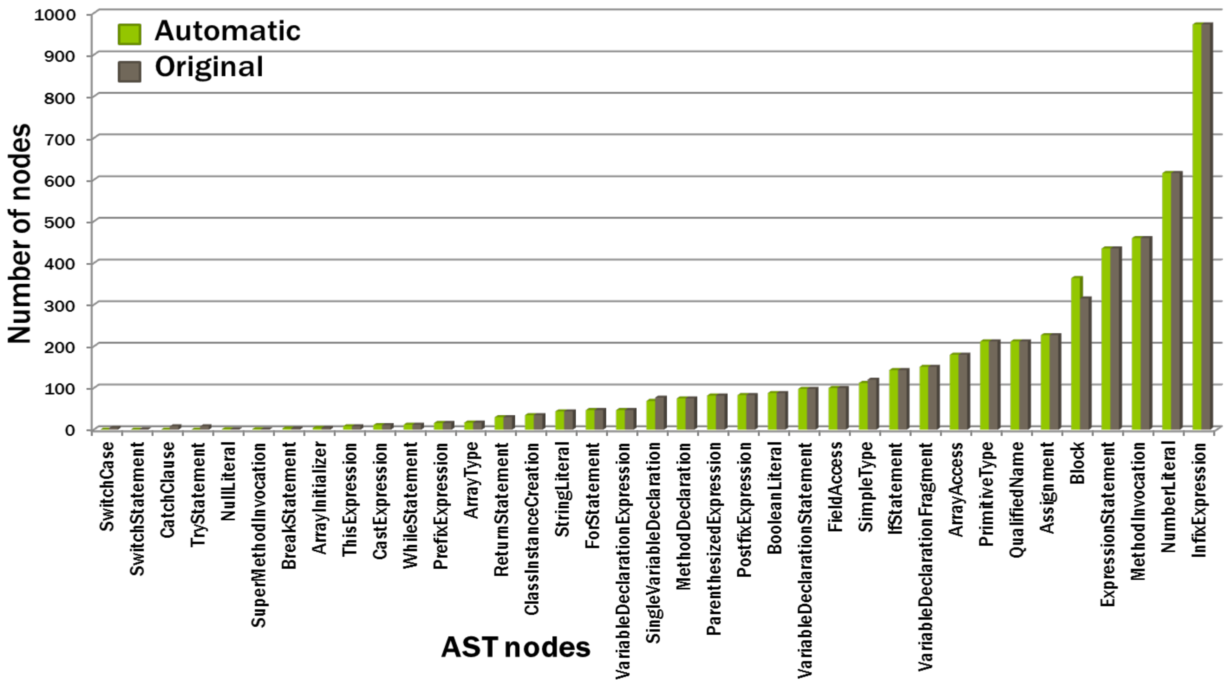
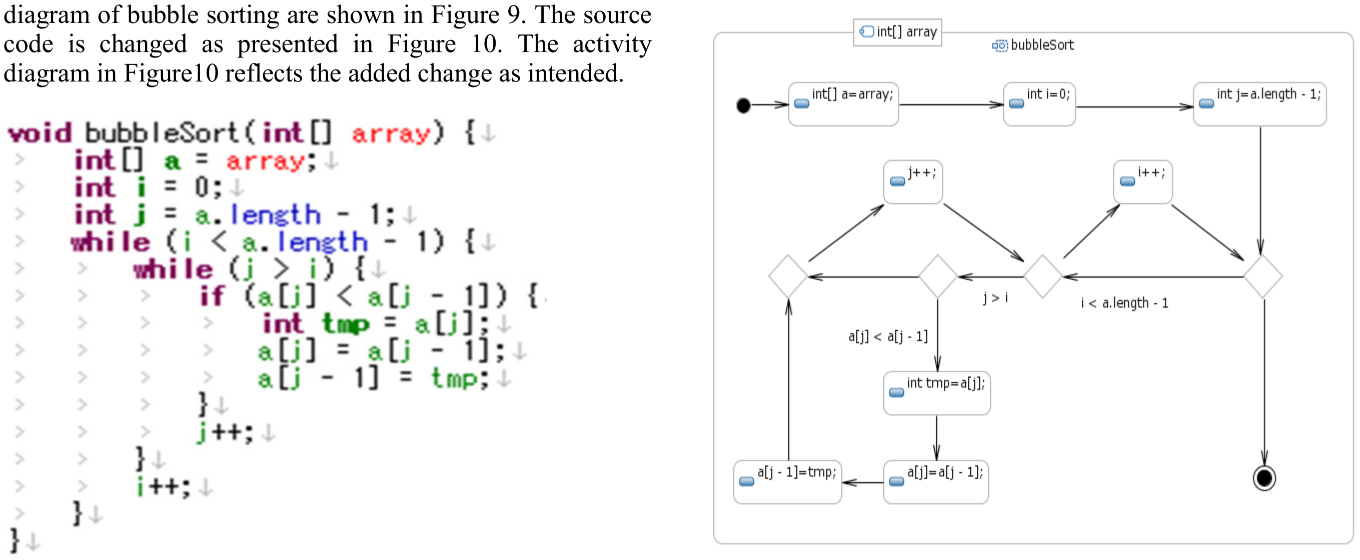


Figure 8. Comparison results of AST nodes.

The transformation rate is 99.6% (= generated XMI nodes * 100 / original XMI nodes). XMI nodes which are not transformed are shown in Table II. There are three kinds 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 decreases in the same number of switch statements in generated activity diagrams. The number of edges is also decreasing in connection with it.

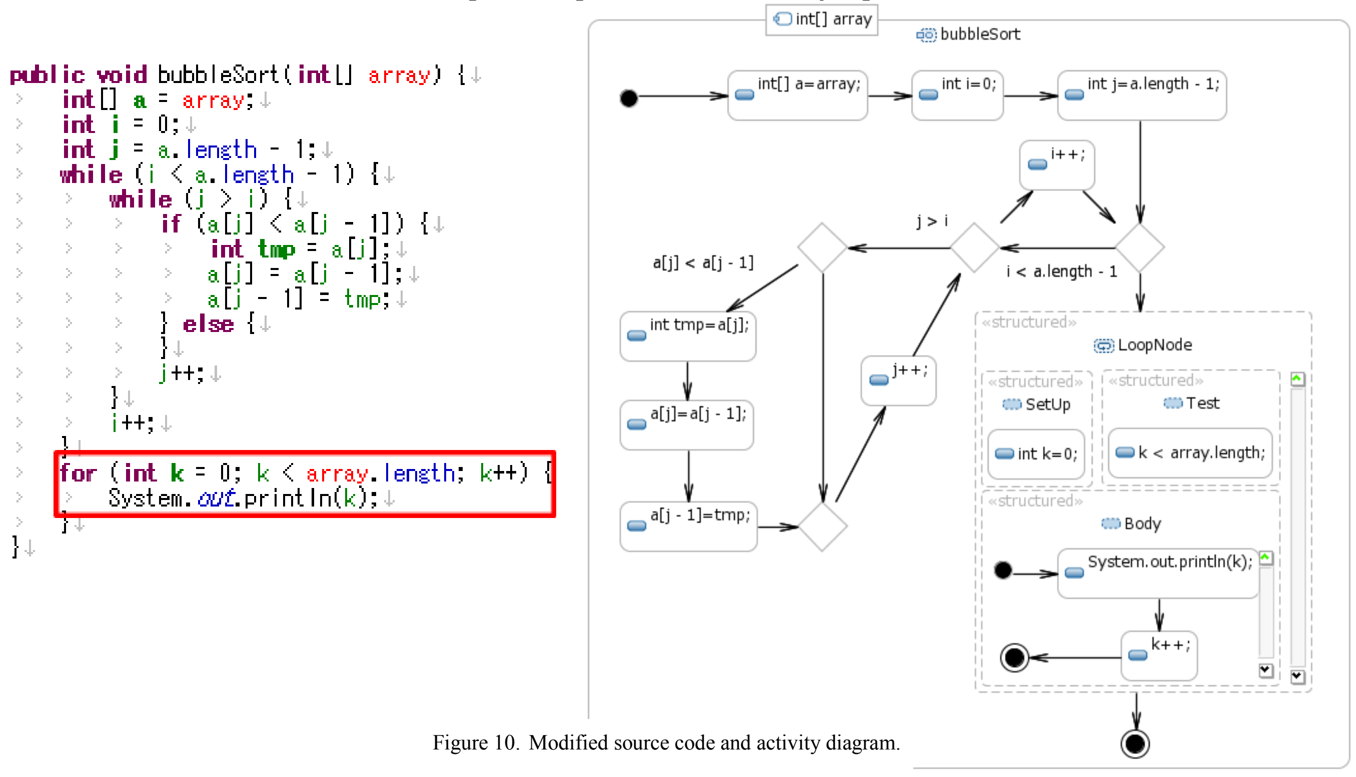
After adding change to source code, an activity diagram is generated from the changed source code. It is verified whether the generated activity diagram reflects the added change. For example, original source code and activity diagram of bubble sorting are shown in Figure 9. The source code is changed as presented in Figure 10. The activity diagram in Figure10 reflects the added change as intended.

A reverse transformation is investigated by generating activity diagrams from handwritten source code and transforming from these activity diagrams to source code. The objects to compare are handwritten source code and the automatic generated source code. The transformation rate is 99.8%. Except for switch statements and the positions of block, they are almost similar. It is verified that the generated source code is functionally equivalent to the handwritten source code. The transformation rates for forward and reverse transformation are not 100% because there is no standard expression to describe switch and try-catch statements in an activity diagram. They are not transformed by the proposed method as shown in Table III.



```
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 9. Original 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 10. Modified source code and activity diagram.

V. CONCLUSION

This paper has pointed out a problem in model driven development and proposed a method of applying round-trip engineering to activity diagrams in order to solve the problem. The effectiveness of the proposed method is verified by the application experiments for the source code of a hunter game. Consequently, it has confirmed that the round-trip engineering between activity diagrams and source code is successful. The characteristics of the activity diagrams accepted by this approach are as follows: They consist of actions of the same granularity, not so many multilayered group nodes.

Since activity diagrams cannot yet deal with switch and try-catch statements, defining of these description methods and increasing convertible elements are important as future work.

ACKNOWLEDGMENT

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

REFERENCES

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

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

[3] A. Uhl, “Model-Driven Development in the Enterprise,” IEEE Software, January/February 2008, pp. 46-49.

[4] R. F. Paige and D. Varró, “Lessons Learned from Building Model-Driven Development Tools,” Software System Model, Vol. 11, 2012, pp.527-539.

[5] 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.

[6] 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.

[7] U. Aßmann, “Automatic Roundtrip Engineering,” Electronic Notes in Theoretical Computer Science, vol. 82, 2003, pp. 33-41.

[8] A. Henriksson and H. Larsson, “A Definition of Round-Trip Engineering,” Technical Report, University of Linköping, Sweden, 2003.

[9] 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.

[10] 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 .

[11] 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.

[12] XML metadata interchange. XMI: [Online]. Available from: http://www.omg.org/spec/XMI/ 2015.3.18.

[13] 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.

[14] E. Gamma, R. Helm, R. Johson, and J. Vlissides, Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley, 1995.

[15] Unified Modeling Language Lab. UML Lab: [Online]. Available from: http://www.uml-lab.com/en/uml-lab/ 2015.3.18.

[16] U. A. Nickel, J. Niere, J. P. Wadsack, and A. Zündorf, “Roundtrip Engineering with Fujaba,” Proc. of the 2nd Workshop on Software-Reengineering (WSR), Bad Honnef, 2000, pp. 1-4.

[17] L. Geiger and A. Zundorf, “Tool Modeling with Fujaba,” Electronic Notes in Theoretical Computer Science, vol. 148, 2006, pp. 173-186.

[18] Acceleo: [Online]. Available from: http://www.eclipse.org/acceleo/ 2015.3.18.

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