Role of Model Transformation in Method Engineering

Motoshi Saeki

Dept. of Computer Science, Tokyo Institute of Technology
Ookayama 2-12-1, Meguro-ku, Tokyo 152, Japan
saeki@cs.titech.ac.jp

Abstract. This paper discusses two applications of model transformation to method engineering; one is method assembly of diagram based methods and formal methods and the other one is providing formal semantics with meta models by means of the transformation of the meta model descriptions into the formal descriptions. We use Class Diagram to define the meta models, and the models following the meta model can be represented with instance graphs. Thus our model transformation is based on graph grammars. To show and clarify the benefits of model transformation in method engineering, we illustrate the transformation rules and how to transform models. We use two examples; one is a method assembly of Class Diagram and Z and the other one is defining formal semantics of the meta model of Class Diagram.

1 Introduction

As information systems to be developed become larger and more complex, modeling methods such as object-oriented analysis and design methods (simply, methods) are being one of the key technologies to develop them with high quality. Modern methods such as RUP[9] adopt the techniques from multiple viewpoints. For example, UML[17], which is used in these methods, has nine diagrams that can depict functional aspects, structure, behavior, collaboration and implementation ones of the system. In this situation, transformation on these models from different views, we call it model transformation, has played an important role as follows;

1. Checking consistency : transforming into a common notation the different descriptions that have overlaps with each other[13]. For example, a data flow diagram and a state diagram in OMT are transformed into logical formulas in order to check if the behavior specified with the state diagram is consistent with the data flow diagram[3].
2. Changing different notations : In Rational Rose, a CASE tool for UML, a sequence diagram can be automatically transformed into a collaboration diagram and vice versa, because these diagrams represent the same view of the system, i.e. collaboration of objects.

A. Banks Pidduck et al. (Eds.): CAISE 2002, LNCS 2348, pp. 626–642, 2002.

3. Providing formal semantics with diagrams : Diagrams such as UML are called semi-formal because they have no rigorous formal semantics in detail level yet. Some studies are related to the techniques of transforming UML diagrams to formal descriptions, e.g. class diagrams to algebraic descriptions[5], state diagrams to LOTOS[22] and so on.
4. Generating a program from a design model : Model descriptions in a design level are translated into an executable model such as a state transition machine. The executable model sometimes is too abstract, and in this case, it is refined further into a more concrete description such as a program by using a stepwise refinement technique adopted in model-based formal description techniques like Z[14] and VDM[11]. In DAIDA project[10], an object-oriented design model of an information system written in TaxisDL language is translated to an abstract state machine which can be refined into an implementation.

In this paper, we discuss two potentials for the other roles of model transformation as the techniques of method engineering for information systems development. According to Sjaak Brinkkemper[7], Method Engineering can be defined as an engineering discipline to investigate how to construct and to adapt methods which are suitable for the situation of development projects. One of the ways to construct suitable methods is method assembly, where we selects method fragments (reusable parts of methods) and assemble them into the suitable methods. In [8,15], method fragments stored in a method base (data base of methods and/or method fragments) are defined as meta models, and method assembly is performed by means of manipulating these meta models. At first this paper suggests another technique for method assembly based on model transformation and this is one of the roles that we will discuss in the paper. Another role is concerned with the way to provide formal semantics with meta models. Some researchers investigated and proposed meta-modeling techniques based on Entity Relationship Model[6], Object-Oriented Model (MOF)[2], Predicate Logic (including Object Z)[18] and attribute grammars[20]. They successfully defined just the structure of the artifacts that were produced in a development project following a method, i.e. *abstract* syntax of the artifacts in a sense, but did not specify the semantics of the artifacts. We apply a model transformation technique to providing formal semantics with meta models.

The rest of the paper is organized as follows. Section 2 sketches the transformation technique based on graph grammars, which we have used in this paper. In section 3, we discuss method assembly of a diagram based method and a formal method, more concretely a class diagram and a formal method Z[14]. In section 4, as another application of model transformation, we discuss the formal semantics of these meta models by providing a set of the transformation rules that can generate a formal description of an instance of a meta model.

2 Graph Rewriting System

A graph rewriting system converts a graph into another graph or a set of graphs following pre-defined rewriting rules. There are several graph rewriting systems such as PROGRESS[19] and AGG[21]. We use the definition of the AGG system. A graph consists of nodes and edges, and type names can be associated with them. Nodes can have attribute values depending on their type. Figure 1(a) is a simple example of rewriting rules. A rule consists of a left-hand (graph B1) and a right-hand (graph B2) which are separated with "::=". A rectangle stands for a node of a graph and it is separated into two parts with a dashed lines. Type name of a node appears in the upper part, while the lower part contains its attribute values. In the figure, the node of "TypeA" in B1 has the attribute "val" and its value is represented with the variable "x". Numerals are used for identifying a node between the left-hand graph and the right-hand graph. For example, the numeral "1" in the node of "TypeA" in B1 means that the node is identical to the node of "TypeA" having "1" in B2. A graph labeled with NAC (Negative Application Condition) appearing in the left-hand controls the application of the rule. If a graph includes the NAC graph, the rule cannot be applied to the graph. The procedure of graph rewriting is as follows ;

1. Extracting the part of the graph that structurally matches to the left-hand of the rule. If the type names are attached with nodes and/or edges, these names should also match during this process. The variables appearing in attributes of the nodes are assigned.
2. Replacing the extracted part with the right-hand of the rule and embedding the result to the original graph, if none of the part which structurally matches a graph of NAC appears. New attribute values are calculated and assigned to the attributes.

Figure 1(b) illustrates rewriting the graph G into the graph H. The triangle part of the graph G is replaced with the rectangular sub graph that is derived from the right-hand of the rule. The attribute values 5 and 2 are assigned to x and y respectively, and those of the two instance nodes of "TypeD" result in 7 (x+y) and 3 (x−y). The other parts of G are not changed in this rewriting process. On the other hand, since the graph I includes the node of "TypeD", the NAC graph of this rule, it cannot be transformed by the rule. In addition, we attach the priority of rule-application with the rules. If we have more than one applicable rule to a graph, we select some of them by their priority and rewrite the graph by using the selected rules at first.

3 Method Assembly of Diagram Based Methods and FDTs

3.1 Method Assembly Based on Transformation

Various kinds of formal description technique (simply, FDT) such as LOTOS, Estelle, SDL, Z[14], and VDM[11] have been developed to specify software systems formally and are putting into practice. The benefits of using FDTs result

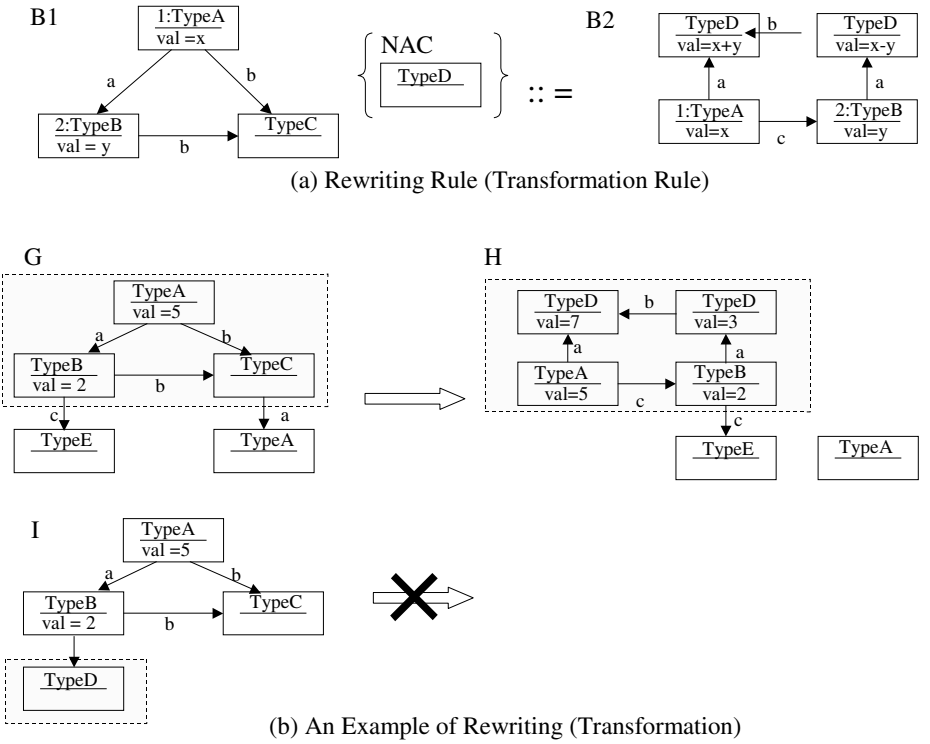

Fig. 1. Graph Rewriting Rule and Rewriting Process

from their rigorous and formal semantics based on mathematics. Software developers can use formal descriptions (FDs) as communication tools, and can verify some properties of them such as consistency and correctness. Furthermore some of the FDs such as LOTOS can be executed as prototype.

However it is difficult for developers to understand documents written in a FDT and to construct a specification in the FDT without learning and training it, because it has specific syntax and semantic rules based on mathematics such as set theory, algebra and mathematical logic. Furthermore no sufficient methods for guiding how to construct formal specifications are embedded to a FDT. Methods such as SA/SD, OMT[16] and RUP[9] guide developers to construct the models of an information system step by step. And almost of them produce diagrams as specifications that are easy for the developers to understand. The methods are one of key factors to efficiently construct the specification of high quality. Method assembly is one of the techniques to combine the methods with FDTs and several case studies have been reported[12,4], even in industries. We assemble an existing method and a FDT into a new method by using transformation rules. More concretely, we transform some artifacts that are developed following the method into the descriptions written in FDT.

Methods are usually used for extracting and identifying an abstract model of the system to be developed, while FDTs are applied to the stage to describe the detailed specification. For example, we compose a class diagram (Object Model), a state transition diagram (Dynamic Model) and a data flow diagram (Function Model) that specify the system from multiple viewpoints, when we adopt the OMT method. These diagrams do not always include detailed or complete descriptions of the system, e.g. detailed contents of the operations on objects do not appear in a class diagram. By this fact, it would be better that we apply these methods at first in the processes of constructing a specification document, and after identifying a model structure of the system i.e. a kind of template, we fill its slots with detailed descriptions of the FDT. To support this process, we design the transformation rules that can generate automatically a template for the FDT descriptions from these diagrams. Thus we can have a new method whose first step is based on the method, e.g. OMT and whose second step is to add FDT descriptions in the artifacts that are produced in the first step. That is to say, we assemble the method with the FDT through the transformation rules.

Our transformation technique should be general in the sense that it can be applied to various kinds of method and FDT. Thus we consider the rules on meta models of methods and FDTs. A meta model represents a method or a FDT itself and, like MOF, we use Class Diagram to describe the meta models. Thus the methods, FDTs and FDs can be mathematically represented with graph. Transformation rules can be defined as graph rewriting rules and the rewriting system can execute the transformation automatically. Furthermore connecting CASE tools for the method with the rewriting system allows us to have an integrated tool supporting seamlessly the processes to construct a FD by using methods.

This process can be summarized as follows;

1. Following the method, we construct a model description, e.g. class diagrams, data flow diagrams, etc., of the system to be developed.
2. From the model description, we get the graph form of the description, called instance graph, based on the meta model of the method.
3. We transform the instance graph into the instance graph for the FDT (i.e. the graph-representation form of a FD) by using the graph rewriting rules.
4. The template of the FD is automatically generated from the FDT instance graph.
5. We describe the detailed parts that are slots in the template.

The details will be discussed in the next section by using an example. Figure 2 sketches the above process.

3.2 Method Assembly Example

In this subsection, we illustrate the method assembly of class diagrams and formal method Z, which is based on ZF set theory and predicate logic. Figure 3 depicts a part of the meta models of class diagrams and of Z.

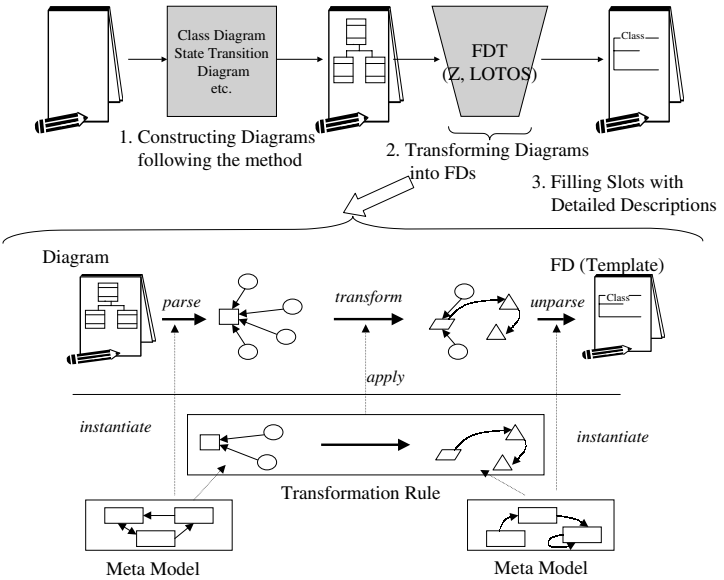

Fig. 2. Method Assembly and Transformation Process

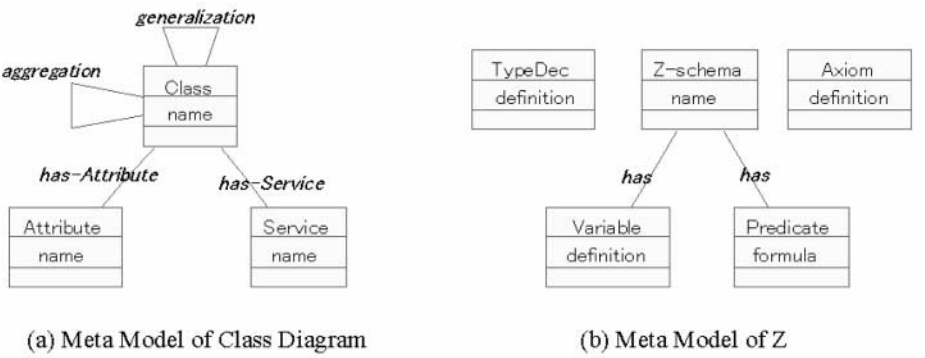

Fig. 3. A Meta Model of Class Diagrams and Z

We must note briefly the conventions on Z notation. The Z schema defines variables and constraints on them with predicate logical formulas in the following style;

TypicalSchema
Signatures (*Variable Declarations*)

Predicate
(*Invariants, and Pre and post condition*)

Figure 4 shows a brief sketch of a class diagram and the Z description which is derived from the class diagram. As shown in (a) of the figure, the class "Lift" in the class diagram, surrounded with a dotted box, is transformed into the Z schema "Lift", considering the associations and the relationships that the "Lift" class participates in. The service "up" in the "Lift" class generates an additional Z schema in order to define the operation "up", as depicted in the right-hand side in the figure. That is to say, a class corresponds to a Z schema, while each service (operation) in a class also does to a Z schema. Attributes and associations are transformed into variables appearing in the Z schema. The Δ notation in the signature part (variable declaration part: the upper of the Z schema) of the service "up" declares the variables whose values may be updated by the operation. The variables with the prime (') decoration represent the values after the operation is executed, while the variables that are not decorated represent the values before the operation. Graph-representation forms of Figure 4(a) following the meta models of Figure 3 (a) are depicted in the figure (b).

Transformation rules of a class diagram into a Z description can be defined as a graph grammar in straightforward way as shown in Figure 5. The rules have the priority, for example, the rule 1) has the priority of 1, the highest one. This rule should be applied at first. After no rules with higher priority are applicable, we can use the rules of the next lower priority. In the figure, we should apply the rule 1), and then do the rules 2), 3) and 4) which have lower priority rather than the rule 1). The priority is useful to easily describe the transformation rules. For simplicity, we do not write down attribute names but just their values included in nodes appearing the rules in the figure, because the readers can identify uniquely the attribute names. Note that these rules derive just a structure of Z schemas based on the class diagram, not strict and formal descriptions of the meaning of the class diagrams. In this sense, the rules produce the template of the Z schemas.

Figure 6 illustrates a series of the snapshots of the transformation process of the example of Figure 4. These are screen dumps of the execution result by AGG graph-rewriting system. See the screen (1) in the figure and it is separated into the top and bottom areas. The top area displays a rule being applied to a graph, which the graph to be rewritten is in the bottom area. Furthermore the top area consists of two or three areas from left to right. These display a NAC, a left-hand side of the rule and a right-hand side respectively. By repeatedly applying the rule 1), for each class appearing in a graph, a node of the corresponding Z

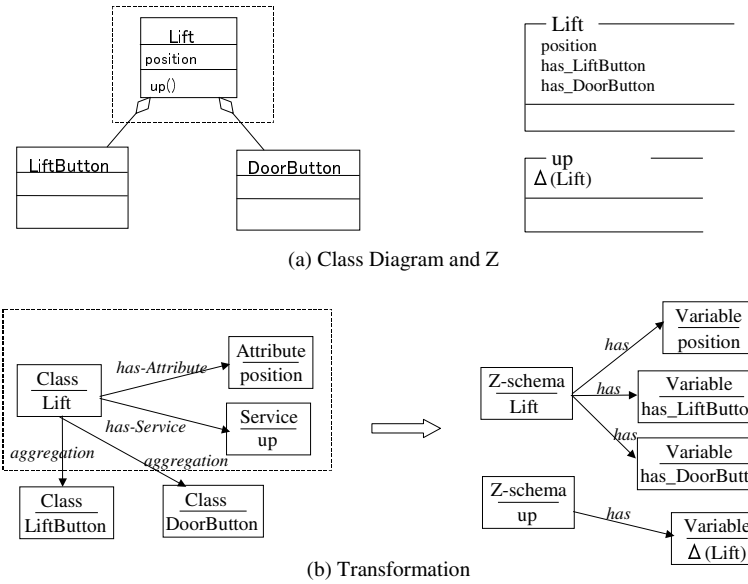

Fig. 4. Transformation of Class Diagrams

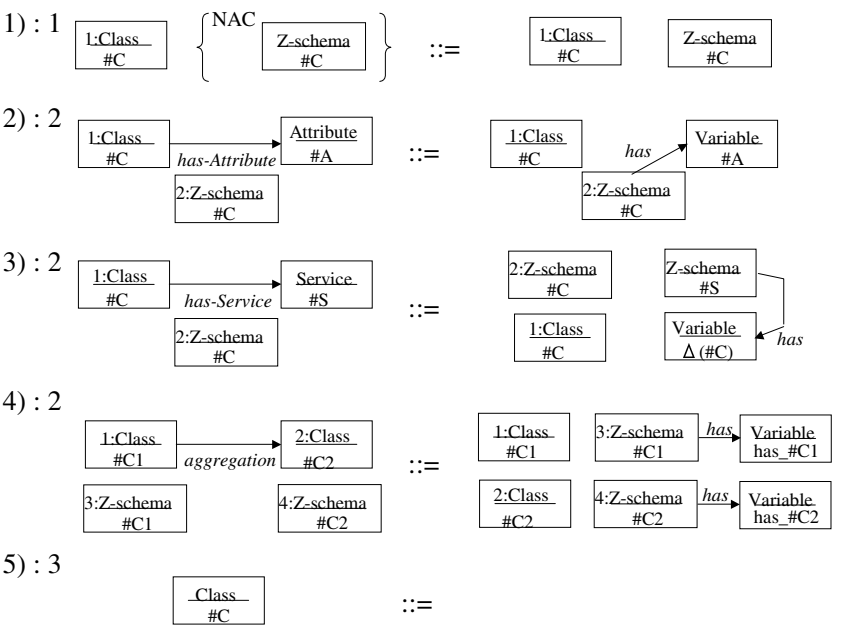

Fig. 5. A Part of Transformation Rules for Method Assembly

schema is generated (the screen (1) in Figure 6). The NAC part of the rule (1) prevents its duplicated application to the same class node. As a result, every class node has the corresponding Z schema node and the rule (1) cannot be applied any more (screen (2)). As shown in the screen (2), we have three Z schemas "Lift", "LiftButton" and "DoorButton". By applying the rule (2), we can produce a node of type "Variable" denoting the attribute "position" and draw an edge from "Lift" node to it. See the transformation from the screen (3) to (4) and the readers can find a new node "Variable" whose the attribute "name" is "position". The rule (4) is for adding a Z schema denoting a service of a class (screen (4)) and its result is shown in the screen (5). Two aggregation relationships to "LiftButton" and "DoorButton" classes are processed by using the rule (4) (screens (5) and (6)). Finally, the class nodes are eliminated with the rule (5) (screen (7)) and we can get a final result as shown in the screen (8).

4 Transformational Semantics of Meta Models

4.1 Overview

Some researchers investigated and proposed meta-modeling techniques based on Entity Relationship Model[8], Predicate Logic (including UML/OCL)[9,18], attribute grammars[20] and MOF[2], in order to define precisely methods. They successfully defined just the structure of the artifacts that were produced in a development project following the method. Consider again the simple example of a meta-model of class diagrams and is depicted in Class Diagram itself, as shown in Figure 3(a). This diagram specifies just the logical structure of class diagrams, i.e. *abstract* syntax in a sense. A crucial question arises in this meta-model description. What is the meaning of this meta-model? More precisely what meaning do "Class" and "generalization" have? One of the answers to the above question is applying Ontology[8]. Ontology consists of a set of atomic words and their relationship structure. The "atomic words" mean the non-decomposable words where persons can have the common understanding. For example, although the word "Class" is not provided with formal semantics, it can be commonly understood by almost all of software engineers. The technique of Ontology is based on constructing a kind of thesaurus for methods and providing a map from method concepts to the atomic words included in the thesaurus. Although this technique is useful to clarify the differences on method concepts between methods, it cannot provide a formal semantics or we cannot have any formal treatment regarding the semantics of meta models.

This section suggests another solution how to provide the meaning for meta-models so that we can treat the semantics of meta models formally. Although the artifacts that are produced following a method such as class diagrams may include informal parts, they can be partially translated into formal descriptions, and the parts that can be translated can have formal semantics. We capture the semantics of a meta-model as the rules that generate the formal specification (Z description) of a real system when we specify the system following the

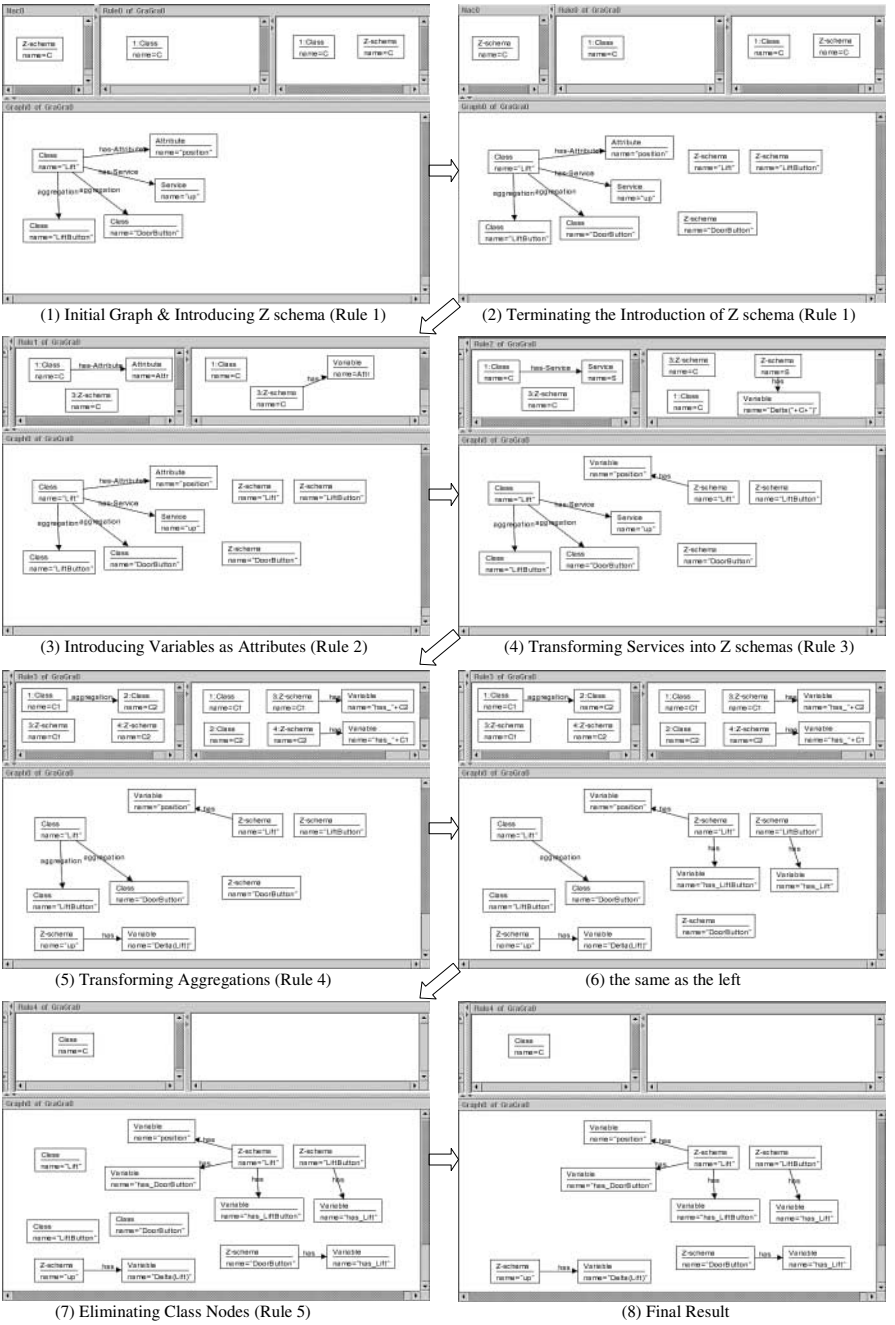

Fig. 6. A Snapshot of A Transformation Process

meta-model. The rules can be considered as model transformation rules and be described in the rules of a graph grammar in the similar way of section 3. Figure 7 sketches the above discussion. A set of the translation rules is associated with each meta-model description as a mechanism which provides its semantics.

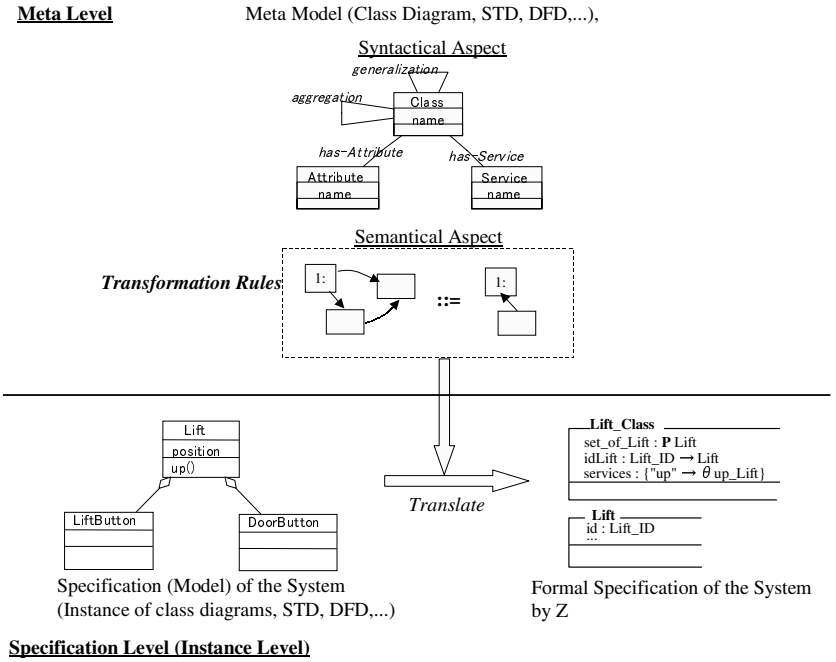

Fig. 7. Semantics of Meta Models by Translation Rules

4.2 Example : Semantics of Class Diagrams by Z

Consider that we write the class diagram not in diagrammatic notation but in formal description technique such as Z. The difficulties in representing a class diagram with Z are how to provide the mechanisms for 1) constructing instances of a class and 2) generalization, i.e. inheritance from a super class. Suppose that a class diagram shown in Figure 8. A class is mathematically a set of objects which has the functions manipulating the existing objects. Adding these functions, we can get the following description of the class "Brake".

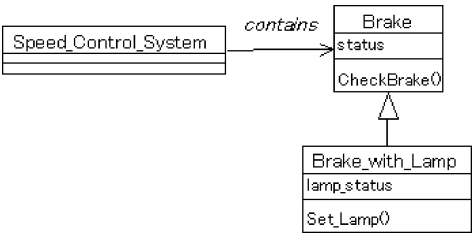

Fig. 8. An Example of A Class Diagram

$[Brake\_ID]$

Brake

$id : Brake\_ID$
$status :$
$contains\_destination : Speed\_Control\_System\_ID$

Brake_Class

$set\_of\_Brake : \mathbb{P}\, Brake$
$idBrake : Brake\_ID \pfun Brake$
$services : \{\text{“}CheckBrake\text{”} \mapsto \theta CheckBrake\_Brake\}$

$idBrake = \{x : set\_of\_Brake \bullet x.id \mapsto x\}$

CheckBrake_Brake

$\Delta(Brake)$
...

$id' = id$
...

Brake_new

$\Delta Brake\_Class$
$new\_Brake! : Brake$

$\neg(new\_Brake!.id \in \mathrm{dom}\, set\_of\_Brake)$
$set\_of\_Brake' = set\_of\_Brake \cup \{new\_Brake!\}$

$[Brake\_with\_Lamp\_ID]$

$Brake\_with\_Lamp\_ID \subseteq Brake\_ID$

Brake_with_Lamp

$Brake \rightarrowtail Brake\_with\_Lamp!$

*Brake_with_Lamp*1

$id : Brake\_with\_Lamp\_ID$
$lamp\_status :$

Brake_with_Lamp_Class

$set\_of\_Brake\_with\_Lamp : \mathbb{P}\, Brake\_with\_L\, amp$
$idBrake\_with\_L\, amp : Brake\_with\_Lamp\_ID \twoheadrightarrow Br\, ake\_with\_L\, amp$
$services : Brake\_Class.servic\, es \leftrightarrow$
$\{\text{“}Set\_L\, amp' \mapsto \theta Set\_L\, amp\_Brake\_with\_L\, amp\}$

$idBrake\_with\_L\, amp =$
$\{x : Br\, ake\_with\_L\, amp \bullet x.id \mapsto x\}$

Brake_with_Lamp_new

$\Delta Brake\_with\_L\, amp\_Class$
$new\_Brake\_with\_Lamp! : Brake\_with\_L\, amp$

$\neg(new\_Br\, ake\_with\_Lamp!.id \in \mathrm{dom}\, set\_of\_Brake\_with\_Lamp)$
$set\_of\_Brake\_with\_Lamp' =$
$set\_of\_Brake\_with\_L\, amp \cup \{new\_Brake\_with\_Lamp!\}$

Set_Lamp_Brake_with_Lamp

Sp eedControl_System

$id : Sp\, eedControl\_System\_ID$
$contains\_sour\, ce : \mathbb{P}\, Br\, ake\_ID$

Sp eedControl_System_Class

The Z schemas “*Brake*” and “*Brake_Class*” define a brake object and its class respectively. The “*Brake_Class*” has the variable “*set_of_Brake*” which holds the information on which brake objects currently exist, i.e. have been already created by its constructor. The map “*idBrake*” in “*Brake_Class*” is for accessing brake objects with their identifiers. Since it is defined as a function, a unique identifier should be attached to a brake object, i.e. their identifiers are different if the objects are different. The operation “*Brake_new*”, a constructor for brake objects, adds a newly generated object “*new_Brake*!” into the set “*set_of_Brake*” and returns it as a result of the operation. The first line of the predicate part of “*Brake_new*” says that the identifier of the newly generated object is not equal to any identifiers of the existing objects. When translating a class diagram into a Z description, we should add the Z schemas for holding the existing objects and for defining constructors, as mentioned above. In addition, the variable “*services*” included in the schema “*Brake_Class*” has the information on which services the schema has. When the name of a service is specified with the variable, say *services* (“*CheckBrake*”), we can get the specification of

the service defined with Z schema. We use the operator θ for specifying the Z schema itself.

Association and aggregation relationships between classes can be formally and simply specified by means of introducing into the Z schemas the variables that hold the relationship information. In the example, we define the variable "*contains_destination*" in the schema "*Brake*", through which the Brake objects are linked to the "*Speed_Control_System*" objects. It is the same technique as introducing the variables like "*has_LiftButton*" in the section 3.2 in order to specify the aggregation relationships.

Inheritance (generalization-specialization) relationships would be more complicated. Mathematically, an object of a sub class is also an object of its super class, and the formula $Brake\_with\_Lamp\_ID \subseteq Brake\_ID$ specifies this mathematical property of set-inclusion. Since the definition of a subclass overrides that of its super class, we use an override operator $\oplus$ on Z schema. Suppose that A is a map. In this case, $(A \oplus \{x \mapsto y\})(z) = y$ if $z = x$, otherwise the resulting value is $A(z)$. In the case that A is a Z schema, $(A \oplus [a : new\_domain]).x$ returns the value of new_domain if $a = x$, otherwise it follows the definition of the schema A. For simplicity, we define the modified version of the override operator $\oplus$ as $A \leftarrowtriangle B = (A \oplus B) \cup B$. The newly defined operator $\leftarrowtriangle$ can play a role on keeping the definitions included in B. By using $\leftarrowtriangle$, we can formally and separately define the override mechanism of attributes and services of classes.

Turn back the example to clarify how to translate inheritance mechanism of super-sub classes. In the example, "*Brake_with_Lamp*" is a sub class of "*Brake*" and its instance has a lamp indicator for notifying pushing the brake. We generate an auxiliary schema "*Brake_with_Lamp*1" which has newly defined variables only, i.e. "*lamp_status*" in addition to its identifier *id*. To get the Z schema of the subclass, we connect this schema to the Z schema of super class with the override operator "$\leftarrowtriangle$", and we can have the schema "*Brake_with_Lamp*" which has both variables of "*Brake*" and of "*Brake_with_Lamp*". It is the definition of the instances of the subclass "*Brake_with_Lamp*". As for the schema "*Brake_with_Lamp_Class*", we just override the variable "*services*" in order to add the services that are newly defined in the sub class.

The above formal description of the class diagram has the strict meaning that is provided by the semantics of Z, i.e. set theory and predicate logic. In other words, if we can have some translation rules of diagrammatic representations to a formal description, the rules define the meaning of the diagrammatic representations. It suggests that the semantics of the meta-models can be defined as the translation rules from the meta-models to Z descriptions, as shown in Figure 7. The rules generate the formal description of the system when the system is specified following a method, i.e. when its meta-model is instantiated with the system specification.

A part of the translation rules of class diagrams into Z as graph rewriting ones can be designed in Figure 9 and these can be considered as formal semantics of the meta model of Figure 3(a).

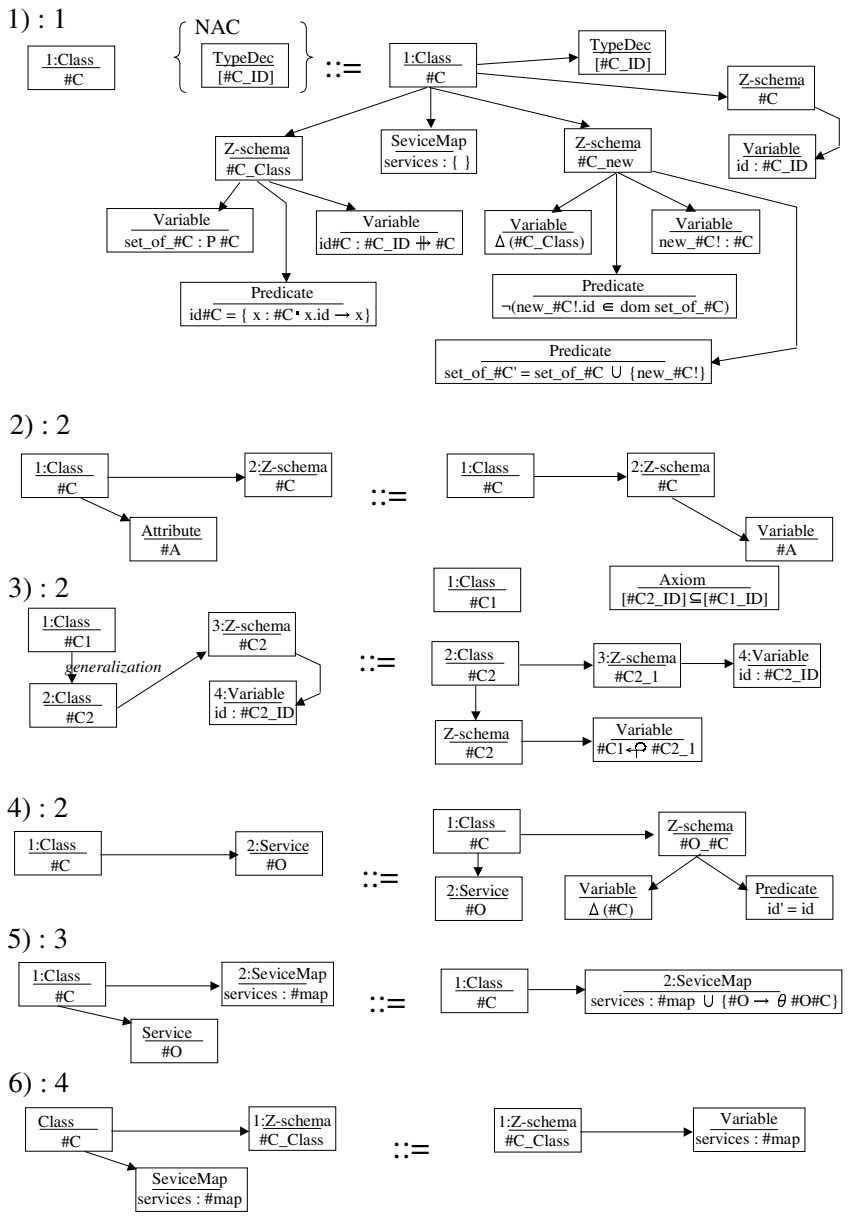

Fig. 9. Transformation Rules for Semantics of Class Diagrams

5 Conclusion and Research Agenda

In this paper, we discuss the roles of model transformation based on meta models in method engineering. The one role that we discussed was transformational approach to method assembly and the other one was formal semantics of meta-model descriptions. The semantics is provided by the rules which transform any model following the meta-model into a formal description. The transformation is defined with a graph grammar and its graph rewriting system can execute automatically the transformation. We also illustrated method assembly of class diagrams and Z, and the formal semantics of the meta model of class diagrams by means of Z. We have developed 12 transformation rules for the method assembly example and 20 rules for providing the example semantics.

In this paper, we have selected Z to provide formal semantics with the artifacts that have been produced following methods. There are several other formal description techniques (FDTs) such as VDM, LOTOS and algebraic specification languages for Abstract Data Type. In particular, some excellent studies to provide formal semantics with the diagrams of OMT exist [5,22]. Although the aim of these researches are the verification and/or consistency checking of specifications written with OMT diagrams, we can apply these techniques to our approach if we select their formal methods such as LOTOS and algebraic languages as a semantic basis on meta-models. We should explore which FDT is suitable for the semantics of meta-models.

The transformation rules are defined with graph grammars. To describe them, as well as meta models and models, with high portability, adopting XML techniques[1] are one of the significant topics. Although the AGG system holds graphs in the form of XML documents, its format is specific and some kind of converter is necessary to combine a method engineering tool.

References

1. XML : eXtensible Markup Language. *ftp://ftp.omg.org/pub/docs/ad/*, 1996. 641
2. Meta Object Facility (MOF) Speicication. *ftp://ftp.omg.org/pub/docs/ad/*, 2000. 627, 634
3. T. Aoki and T. Katayama. Unification and Consistency Verification of Object-Oriented Analysis Models. In *Proc. of 5th Asia-Pacific Softwrae Engineering Conference (APSEC'98)*, pages 296–303, 1997. 626
4. D. Berry and M. Weber. A Pragmatic, Rigorous Integration of Structural and Behavioral Modeling Notations. In *Proc. of 1st International Conference on Formal Engineering Methods*, pages 38–48, 1997. 629
5. R. Bourdeau and B. Cheng. A Formal Semantics for Object Model Diagrams. *IEEE Trans. on Software Engineering*, 21(10):799 – 821, 1995. 627, 641
6. S. Brinkkemper. *Formalisation of Information Systems Modelling.* Thesis Publisher, 1990. 627
7. S. Brinkkemper. Method Engineering : Engineering of Information Systems Development Methods and Tools. *Information and Software Technology*, 37(11), 1995. 627

8. S. Brinkkemper, M. Saeki, and F. Harmsen. Meta-Modelling Based Assembly Techniques for Situational Method Engineering. *Information Systems*, 24(3):209–228, 1999. 627, 634
9. I. Jacobson, G. Booch, and J. Rumbaugh. *The Unified Software Development Process*. Addison Wesley, 1999. 626, 629, 634
10. M. Jarke, J. Mylopoulos, J. Schmidt, and Y. Vassiliou. DAIDA : An Environment for Evolving Information Systems. *ACM Trans. on Information Systems*, 10(1):1–50, 1992. 627
11. C. B. Jones. *Systematic Software Development Using VDM*. Prentice Hall, 1986. 627, 628
12. K. Kronlöf, editor. *Method Integration – Concepts and Case Studies*. Wiley, 1993. 629
13. C. Pons, R. Giandini, and G. Baum. Dependency Relations Between Models in the Unified Process. In *Proc. of 10th International Workshop on Software Specification and Design (IWSSD-10)*, pages 149–158, 2000. 626
14. B. Potter, J. Sinclair, and D. Till. *An Introduction to Formal Specification and Z*. Prentice Hall, 1996. 627, 628
15. J. Ralyte and C. Rolland. An Assembly Process Model for Method Engineering. In *Lecture Notes in Comupter Science (CAiSE'01)*, volume 1626, pages 267–283, 2001. 627
16. J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lonrensen. *Object-Oriented Modeling and Design*. Prentice-Hall, 1991. 629
17. J. Rumbaugh, I. Jacobson, and G. Booch. *The Unified Modeling Language Reference Manual*. Addison Wesley, 1999. 626
18. M. Saeki and K. Wenyin. Specifying Software Specification & Design Methods. In *Lecture Notes in Computer Science (CAiSE'94)*, pages 353–366. Springer-Verlag, 1994. 627, 634
19. A. Schurr. Developing Graphical (Software Engineering) Tools with PROGRES. In *Proc. of 19th International Conference on Software Engineering (ICSE'97)*, pages 618–619, 1997. 628
20. X. Song and L. J. Osterweil. Experience with an Approach to Comparing Software Design Methodologies. *IEEE Trans. on Soft. Eng.*, 20(5):364–384, 1994. 627, 634
21. G. Taentzer, O. Runge, B. Melamed, M. Rudorf, T. Schultzke, and S. Gruner. AGG : The Attributed Graph Grammar System. *http://tfs.cs.tu-berlin.de/agg/*, 2001. 628
22. E. Wang, H. Richer, and B. Cheng. Formalizing and Integrating the Dynamic Model within OMT*. In *Proc. of 19th International Conference on Software Engineering*, pages 45 – 55, 1997. 627, 641