Specifying Software Specification & Design Methods

Motoshi Saeki and Kuo Wenyin

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

Abstract. To support customizing and integrating software specification & design methods to a suitable method for designers' problem domain and environment, so-called Computer Aided Method Engineering (CAME), we need a meta model for representing the fragments of methods formally and for composing them into a method. This paper discusses a meta modelling technique by using a formal specification language Object-Z which is an object oriented version of the Z language. The logical expressions of Object-Z allows us to describe hierarchical structures and the constraints in the methods and the inheritance mechanism enables us to integrate method fragments into a new method.

1 Introduction

It is important to design software specifications effectively for developing high quality software with low cost because specification & design phases are the early step in the software development process. Many specification & design methods (shortened to methods) such as Structured Analysis & Design [5] and Object-Oriented Analysis & Design [12] have been developed to guide designers' work. However, these methods can work well only in some problem domains and/or environment, not in all, and it is very difficult to create an universal method which can work well in all the domains and/or environment. It is more feasible that the designers can select suitable methods, customize, and integrate the methods to a suitable one for their problem domains and environment.

Recently, there are some methods with multiple viewpoints to develop the large and complex software systems. For example, Shlaer's and Mellor's Object Oriented Analysis[12] can be considered as a *multi-view* method since its underlying model is the composition of three models — an information model (Entity Relationship Diagram), a state model (State Transition Diagram) and a process model (Data Flow Diagram). We know that the specifications described in several methods with the different viewpoints are more useful[4]. However, it is a problem how to integrate the specifications developed by using the different methods into a final specification. To support *multi-view specification*, we also need a mechanism for integrating the specifications written in the different methods.

One of the possible solutions for the above requirements is to use a *meta system* or *meta model* approach[1] for method modelling. The meta model is a data model or scheme for representing methods, and expresses a common conceptual structure for them. Most of the meta models which have been studied until recently are based on Entity Relationship model (shortened to ER model)[7, 14, 15]. ER model allows us to represent the methods comprehensively, but it is difficult to describe the constraints and the hierarchical structures of the methods. Most of the meta model approaches except for [2] did not deal with the constraints or the hierarchical structure. Knuth's attribute grammer approach[9] could be used to represent the hierarchical structure of products produced in the methods. However, it should include many evaluation rules called *copy rules* to specify any systems and the many occurrences of these non-essential rules allow us to construct the incomprehensible descriptions. Our technique is based on the formal specification language Object-Z[6] to specify the constraints comprehensively. Object-Z is an object oriented extension of a Z language[16] and its notation is the same as that of Z. Hierarchical structures can also be represented with *mathematical maps* or *relations* in Object-Z. Furthermore the inheritance mechanism of Object-Z allows us to integrate methods into one. Object oriented paradigm provides the reusability of method fragment descriptions for constructing new methods.

The organization of this paper is as follows. In the next section, we discuss two kinds of method modelling techniques — one is based on ER model and another is on attribute grammers. We introduce our method modelling technique based on Object-Z language in section 3. The class of data flow diagrams is also specified in our framework as an example here. Section 4 presents our meta model application — method integration. We pick up Shlaer and Mellor's OOA as an example and its description can be obtained from the four popular methods ; Data Flow Diagram, Entity Relationship Diagram, Object Communication Diagram, State Transition Diagram. The constraints for integrating these diagrams can be represented in our technique. These examples show that our modelling technique is sufficiently powerful in expression, and suitable to be a basis for Computer Aided Method Engineering.

2 Method Modelling based on ER Model and Attribute Grammer

2.1 Method Modelling based on ER model

Many studies on meta models based on ER model have been done, i.e. they used ER modeling technique to represent methods. As a simple example, let's consider the definition of data flow diagrams of Structured Analysis by using ER model. The definition has the entities for the nodes of data flow diagrams such as Processes (Bubbles) and Data Flows, and relationships between the entities for the edges or connections among the nodes, as shown in Figure 1. ER model could represent the various methods widely and easily. However, this figure does not express a *well-formed* data flow diagrams completely. In a well-formed diagram it is not permitted to directly connect the data stores to the source&sinks with data flows, but the figure 1 does not contain this constraint.

There is one more shortcoming. ER model cannot express the hierarchical structures of data flow diagrams and the constraints of the hierarchy used in

Structured Analysis. For example, a process in a data flow diagram can' be hierarchically decomposed and refined to another data flow diagram. That is to say, the inside of the process is a lower-level data flow diagram. In this hierarchical structure, the inputs and outputs of the process should be equal to the external inputs and outputs of its lower-level data flow diagram.

It is difficult to represent these kinds of hierarchical structure and constraints as mentioned above by using ER model, even though ER model is such a simple vehicle to describe the methods.

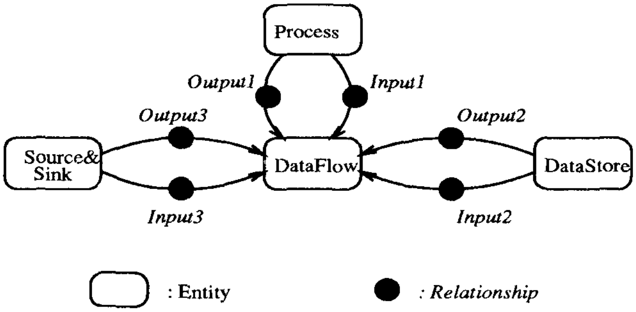


Fig. 1. A Definition of Data Flow Diagram in ER Model

2.2 Method Modelling based on Attribute Grammers

Attribute grammer approach can be an alternative to define the hierarchies and the constraints discussed before. It is an extension of context-free grammers and was proposed by Knuth to specify the formal semantics of programming languages. Attribute grammer based language was used to describe software processes[8], but unlike this, we apply an attribute grammer approach to specifying products such as data flow diagrams.

An attribute grammer consists of a set of the derivation rules associated with the evaluation rules and the conditions. The evaluation rules are used to calculate the attribute values associated with grammatical symbols in the derivation rules. The condition expresses a constraint that must be satisfied by the attribute values when the derivation rule is applied. That is to say, we cannot apply the derivation rules whose conditions do not hold.

Consider the definition of the data flow diagram in attribute grammer approach. The four entities – Process, DataStore, Source&Sink and DataFlow, and the six relationships in Figure 1 can correspond to non-terminal symbols which have the set of entity instances or relationship instances as their attributes. The first derivation rule specifies that data flow diagrams consists these ten components denoted by the non-terminal symbols. We associate a synthesized attribute "product" with the non-terminal symbols. The value of the attribute "product" is a set of the instances of the entities or relationships belonging to the non-terminal symbols, i.e. product itself. For example," product(<Process>)" denotes the set of instances of the processes in the data flow diagram. The conditions $Condition_1$ and $Condition_2$ express that both "Input1" and "Output1" are the relationships between "Process" and "DataFlow". The attribute value "process_role(<Input1>)" denotes the set of processes participating in the relationship "Input1". As you can find in the derivation rule of "Input1", this relationship is defined as a pair of a process and a data flow which is an input to the process. For the other relationships such as "Input2", "Input3" and so on, we can define the similar conditions. *Condition_3* in the derivation rule of <DataFlowDiagram> specifies that neither data stores nor source&sinks can connect directly with each other through any data flows. We can derive a data flow diagram by this rule if all of the conditions attached with it are satisfied, i.e. well-formed data flow diagrams should necessarily meet the conditions.

```
<DataFlowDiagram>::= <Process> <DataFlow> <DataStore> <Source&Sink>
          <Input1> <Input2> <Input3> <Output1> <Output2> <Output3>
      product(<DataFlowDiagram>) ← ···
      Condition_1 : (process_role(<Input1>) ∪ process_role(<Output1>))
                                        = product(<Process>)
      Condition_2 : (dataflow_role(<Input1>) ∪ dataflow_role(<Output1>)
                                        = product(<DataFlow>)
      ...
      Condition_3 : (dataflow_role(<Input2>) ∪ dataflow_role(<Input3>)) ∩
            (dataflow_role(<Output2>) ∪ dataflow_role(<Output3>)) = ∅
<Process>::= ε
   | <process_instance> <Process>_2
      product(<Process>) ← {id(<process_instance>)} ∪ domain(<Process>_2)
<DataFlow>::= ···
      product(<DataFlow>) ← ···
...
<Input1>::= ε
      process_role(<Input1>) ← ∅
      dataflow_role(<Input1>) ← ∅
   | ( <process_instance> , <dataflow_instance> ) <Input1>_2
      product(<Input1>) ← { (product(<process_instance>, <dataflow_instance>) }
                  ∪ product(<Input>_2)
      process_role(<Input1>)
         ← { product(<process_instance>) } ∪ product_role(<Input1>_2)
      dataflow_role(<Input1>)
         ← { product(<dataflow_instance> } ∪ dataflow_role(<Input1>_2)
<Input2>::= ···
...
<process_instance>::= <identifier>
      product(<process_instance>) ← Sring(<identifier>)
...
```

We can also express the hierarchical structure of the data flow diagrams by adding the following derivation rule to the above rules.

```
<process_instance>::= <DataFlowDiagram>
```

The constraints which must be satisfied by the refined process instance <process_instance> and its lower-level data flow diagram <DataFlowDiagram> can be also specified in this attribute grammer approach. To specify them, we should add more attributes, evaluation rules, and conditions to the above grammer, and omit them on account of space.

The method model based on attribute grammers can express most of the methods and solve the problems in ER model. However, one of their shortcomings is that we need a lot evaluation rules such as value copy rules. As shown in the example of DataFlowDiagram above, we must also introduce many conditions for representing such a scheme of the data flow diagram as Figure 1. Many rules and conditions might fail down in incomprehensible descriptions of the methods.

3 Method Modelling based on Object-Z

The formal specification language Object-Z is an object oriented extension of the Z language semantically based on ZF set theory. In object oriented paradigm, the system to be specified is considered as a collection of individual objects having internal states. Object-Z defines the objects by using class concepts where the definition of their states, initial states, and the operations related to them are encapsulated. The class schema for the specification of a class may contain several kinds of schemas as well as the definitions of axioms, predicates, types, and constants. The typical class schema is be shown in the following:

```
┌─ Typical_Class_Schema ─────────────────────────────
│ InheritedClasses
│ ┌──────────────────────────────────────────────
│ │ StateVariableDeclaration
│ ├──────────
│ │ StateInvariants
│ └──────────────────────────────────────────────
│
│ ┌─ INIT ───────────────────────────────────────
│ │ InitialState
│ └──────────────────────────────────────────────
│
│ ┌─ OperationSchema ────────────────────────────
│ │ Signatures ( VariableDeclarations)
│ ├──────────
│ │ Predicate
│ │ (Pre and post condition)
│ └──────────────────────────────────────────────
│ ...
└────────────────────────────────────────────────────
```

The inherited classes are the names of the super classes whose states and operations are inherited to the class *Typical_Class_Schema*.

The aim of the methods is the navigation of designers' activities to develop specifications. The methods tell the designers what documents they should produce in a specification process, and what activities they should perform for producing the documents. So we can model the methods from two perspectives — product and activity perspectives. From the product perspective, the structures or types of the produced products (incl. hierarchical structures) and constraints on the product parts should be specified to define the method. To specify the activities in the method, we define permitted manipulations on its products and their behavioral constraints such as execution ordering. We describe a product specification of the method by a class schema in Object-Z since the instances of the produced products can be considered as objects in object oriented paradigm. The product structure and the constraints can be specified by a state schema. Operation schemas encapsulated in a class schema define the manipulations on the corresponding product, and the pre- and post-conditions of the defined operations specify the behavioral constraints on them such as possible execution order.

We begin with a simple example of the specification of the class *Product* written in Object-Z. It will be used as a super class to specify the product classes of the various methods.

Product

$title : Identifier$
$version : Version\_Number$
$status : Status$
$creation\_date : Date$
$last\_modification\_date : Date$
$producer : Person$
$responsible\_person : Person$
$reviewers : \mathbb{P}\, Person$

INIT

$designer?, manager? : Person$

$version = 0$
$status = `in\_progress'$
$producer = designer?$
$responsible\_person = manager?$

Notify_Completion

$\Delta(status, creation\_date)$
$today? : Date$

$status = `in\_progress'$
$status' = `completion'$
$creation\_date = today?$

Review

$\Delta(status, reviewers)$
$review\_team? : \mathbb{P}\, Person$
$review\_result! : Review\_Report$

$status = `completion'$
$status' = `reviewing'$
$reviewers' = review\_team?$

...

An object of the class *Product* has several state variables such as *title*, *version*, *status* and so on.`Assume that the domains of these state variables, e.g. *Identifier*, *Date* and *Person*, would be externally defined as basic types. The operation *Notify_Completion* sets up the value of the state *creation_date* when the development of the current version of the product is completed. We must note the conventions on variables used in the operation schema. The Δ notation in the signature part declares the variables whose values may be updated by the operation. The state variables with the prime (') decoration represent the state after the operation, while the variables which are not decorated represent the state before the operation. Inputs and outputs of the operation are denoted by the variables with "?" and "!" respectively. In the schema *Notify_Completion*, 'today?' is an input to this operation and the status is also changed from 'in_progress' to 'completion' after this operation

We define the generic schema *ConnectedGraph* before the next class *DataFlow-Diagram.* This schema specifies the constraints on a special class of directed graphs whose nodes have at least one connected edge, i.e. an input edge or an output edge.

ConnectedGraph [*Nodes*, *Edges*, *InputEdges*, *OutputEdges*]

$$\begin{array}{l} Nodes = (\operatorname{ran} InputEdges \cup \operatorname{ran} OutputEdges) \land \\ Edges = \operatorname{dom} InputEdges = \operatorname{dom} OutputEdges \end{array}$$

Several Object-Z operators on sets and relations occur in the logical formulas of the schema. The domain and range operators, dom and ran, extract the domain and range of a relation or a function respectively, i.e. $\operatorname{dom} R = \{x \mid (x, y) \in R\}$ and $\operatorname{ran} R = \{y \mid (x, y) \in R\}$ where R is a relation or a function. *Nodes* and *Edges* are certain sets of nodes and edges respectively, and *InputEdges* and *OutputEdges* denote relationships between *Nodes* and *Edges*. For example, *InputEdges* expresses which nodes the edges are inputs to. Assume that these relationships are defined as finite functions $Edges ⇻ Nodes$. Thus the term "ran *InputEdges*" denotes a set of nodes to which there is at least one input edge.

We can specify a class schema for data flow diagrams and the operations on them by using *Product* and *ConnectedGraph* as shown in the next page. The class *DataFlowDiagram* incorporates all the features such as state variables, invariants, and operations of the *Product* class. For example, *DataFlowDiagram* has the state variables *title*, *status*, the operations *Notify_Completion*, *Review* and so on. In addition, the state variables or structural components of the class contain four sets (corresponding to entities in ER model) and six finite functions (corresponding to relationships in ER model), and this definition comes from Figure 1. *Process*, *DataFlow*, *DataStore*, and *Source&Sink*, which are used for defining domains of the states, are considered as basic types which are externally-given sets. The operator $\mathbb{P}$ stands for the power set. For example, the domain of the *processes* of *DataFlowDiagram* is a power set of the given set *Process*. The relationships between *DataFlow* and other entities are defined as functions because these are one-to-many relationships, i.e. each data flow has just one source and just one destination. When the development of a data flow diagram is completed, it should meet the constraint *WellFormedDataFlowDiagram* which is specified in the axiomatic definition below the state schema. It consists of two logical conjuncts — the first one specifies that processes, source&sinks, and data stores in a data flow diagram should have at least one data flow as their input or output. In addition, source&sinks and data stores can be connected only to processes through data flows. It means that there are no data flows directly between a source&sink and a data store, or data stores. The second conjunct stands for this constraint.

The *DataFlowDiagram* class has several operations on its instances. The operation *IdentifyProcesses* corresponds to the designers' activities for identifying processes and it adds a newly identified process to the state variable *processes*. The second operation *IdentifyInputs*, which corresponds to the activities for identifying an input data flow to a certain process, cannot be performed until the process has been already identified. The first logical formula on the variable "*process?* in the predicate part specifies this behavioral constraints. That is to say, this operation can be performed after at least one execution of the operation *IdentifyProcesses*. By using the predicates in operation schemas, we can specify the behavioral constraints such as execution order on the activities.

DataFlowDiagram

Product

$processes : \mathbb{P}\, Process$
$dataflows : \mathbb{P}\, DataFlow$
$datastores : \mathbb{P}\, DataStore$
$source\&sinks : \mathbb{P}\, Source\&Sink$
$input1 : DataFlow \mathrel{+\!\!\!\twoheadrightarrow} Process$
$output1 : DataFlow \mathrel{+\!\!\!\twoheadrightarrow} Process$
$input2 : DataFlow \mathrel{+\!\!\!\twoheadrightarrow} Source\&Sink$
$output2 : DataFlow \mathrel{+\!\!\!\twoheadrightarrow} Source\&Sink$
$input3 : DataFlow \mathrel{+\!\!\!\twoheadrightarrow} DataStore$
$output3 : DataFlow \mathrel{+\!\!\!\twoheadrightarrow} DataStore$

$status = \text{‘}completion\text{’} \Rightarrow WellFormedDataFlowDiagram$

$WellFormedDataFlowDiagram : \mathbb{B}$

$$\begin{aligned}
&WellFormedDataFlowDiagram = \\
&\quad ConnectedGraph[processes \cup source\&sinks \cup datastores, \\
&\qquad input1 \cup input2 \cup input3, output1 \cup output2 \cup output3] \land \\
&\quad (\operatorname{dom} input2 \cup \operatorname{dom} input3)) \cap (\operatorname{dom} output2 \cup \operatorname{dom} output3) = \varnothing
\end{aligned}$$

IdentifyProcesses

$\Delta(processes)$
$new\_process? : Process$

$processes' = processes \cup \{new\_process?\}$

IdentifyInputs

$\Delta(dataflows, input1)$
$process? : Process$
$new\_dflow? : DataFlow$

$process? \in processes$
$dataflows' = dataflows \cup \{new\_dflow?\}$
$input1' = input1 \cup \{new\_dflow? \mapsto process\}$

...

To define hierarchical data flow diagrams, we introduce a function from processes to lower-level data flow diagrams. This function denotes what data flow diagram a process is refined to. The class of hierarchical data flow diagrams can be recursively defined by using the inheritance from the class *DataFlowDiagram*.

The class schema *HierarchicalDataFlowDiagram* has the axiomatic definition *WellFormedHierarchicalDataFlowDiagram* which defines a constraint for preserving consistency on input-output data flows between a refined process and its lower level data flow diagram. In other words, the input flows and output ones of the refined process should be equal to inputs and outputs between the lower-level data flow diagram and the external environment. In the definition of *WellFormedHierarchicalDataFlowDiagram*, you will find the operator $\rhd$ called range restriction. It reduces a relation or function to one which has a given range, e.g. we have $input1 \rhd \{p\} = \{dfd \mapsto p \mid input1(dfd) = p\}$ where $p \in Process$, $dfd \in DataFlow$, and $input1$ is a state variable of *DataFlowDiagram* class.

$$
\begin{array}{l}
\textit{HierarchicalDataFlowDiagram} \\
\hline
\textit{DataFlowDiagram} \\
\quad \begin{array}{l}
\hline
\textit{refine} : \textit{Process} \twoheadrightarrow\!\!\!\!\!\!+\; \textit{HierarchicalDataFlowDiagram} \\
\hline
\textit{status} = \text{`}\textit{completion}\text{'} \\
\quad \Rightarrow \textit{WellFormedHierarchicalDataFlowDiagram} \\
\hline
\end{array} \\
\quad \begin{array}{l}
\textit{WellFormedHierarchicalDataFlowDiagram} : \mathbb{B} \\
\hline
\textit{WellFormedHierarchicalDataFlowDiagram} = \\
\quad \operatorname{dom} \textit{refine} \subseteq \textit{processes} \land \\
\quad \forall p : \operatorname{dom} \textit{refine} \bullet (\operatorname{dom}(\textit{input}1 \rhd \{p\}) = \textit{InputFlows}(\textit{refine}(p)) \land \\
\qquad\qquad\qquad\qquad \operatorname{dom}(\textit{output}1 \rhd \{p\}) = \textit{OutputFlows}(\textit{refine}(p)))
\end{array} \\
\quad \begin{array}{l}
\textit{InputFlows} : \textit{HierarchicalDataFlowDiagram} \rightarrow \textit{DataFlow} \\
\hline
\forall \textit{hdfd} : \textit{HierarchicalDataFlowDiagram} \bullet \textit{InputFlows}(\textit{hdfd}) = \\
\quad \operatorname{dom}(\textit{input}1.\textit{hdfd} \cup \textit{input}2.\textit{hdfd} \cup \textit{input}3.\textit{hdfd}) \\
\qquad \setminus \operatorname{dom}(\textit{output}1.\textit{hdfd} \cup \textit{output}2.\textit{hdfd} \cup \textit{output}3.\textit{hdfd})
\end{array} \\
\quad \begin{array}{l}
\textit{OutputFlows} : \textit{HierarchicalDataFlowDiagram} \rightarrow \textit{DataFlow} \\
\hline
\forall \textit{hdfd} : \textit{HierarchicalDataFlowDiagram} \bullet \textit{OutputFlows}(\textit{hdfd}) = \\
\quad \operatorname{dom}(\textit{input}1.\textit{hdfd} \cup \textit{input}2.\textit{hdfd} \cup \textit{input}3.\textit{hdfd}) \\
\qquad \setminus \operatorname{dom}(\textit{output}1.\textit{hdfd} \cup \textit{output}2.\textit{hdfd} \cup \textit{output}3.\textit{hdfd})
\end{array} \\
\quad \begin{array}{l}
\textit{RefineProcesses} \\
\hline
\Delta(\textit{refine}) \\
\textit{refined\_process?} : \textit{Process} \\
\textit{lowerdfd!} : \textit{HierarchicalDataFlowDiagram} \\
\hline
\textit{refined\_process?} \in \textit{processes} \\
\textit{refined\_process?} \notin \operatorname{dom} \textit{refine} \\
\textit{title}.\textit{lowerdfd!} = \textit{refined\_process?} \\
\textit{refine}' = \textit{refine} \cup \{\textit{refined\_process?} \mapsto \textit{lowerdfd!}\} \\
\hline
\end{array} \\
\quad \ldots \\
\hline
\end{array}
$$

The functions *InputFlows* and *OutputFlows*, which are used in *WellFormedHierarchicalDataFlowDiagram*, calculate a set of the input data flows from the external environment and a set of the output data flows to the external respectively. The term *input1.hdfd* occurring in the definitions *InputFlows* and *OutputFlows* denotes the value of the state variable *input1* of the data flow diagram *hdfd*, i.e. the relationship between processes and their input data flows in *hdfd*. The operator \, appearing in the definitions stands for set difference, i.e. {a,b,c}\{b,d} is equal to {a,c}. This operator in the definition "InputFlows" calculates the data flows which are inputs to some processes, data stores or source&sinks ($\operatorname{dom}(\textit{input}1 \cup \textit{input}2 \cup \textit{input}3)$) but which has no relation to anything as outputs ($\operatorname{dom}(\textit{output}1 \cup \textit{output}2 \cup \textit{output}3)$).

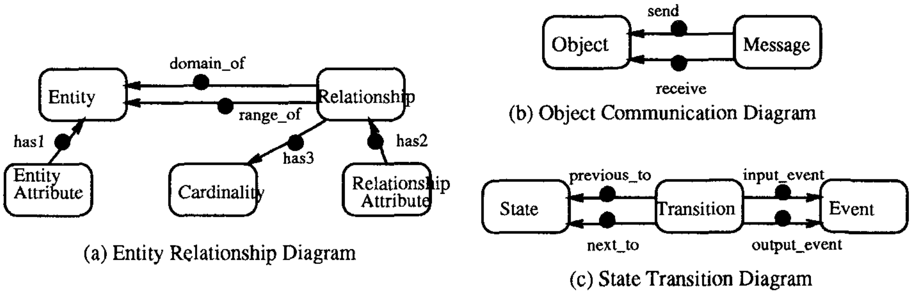
The operation "RefineProcess" is newly added and it denotes the activities for constructing a data flow diagram (*lowerdfd!*) of a process (*refined_process?*). All of the operations defined in *DataFlowDiagram* can be applied to the instances of *HierarchicalDataFlowDiagram*.

As shown in this section, Object-Z language has powerful constructs for defining hierarchical data structures and for specifying constraints comprehensively. It can be considered as one of suitable techniques for specifying not only software specifications but also specification and design methods.

4 Method Integration – An Example

The previous section have presented the advantages of Object-Z language to use method descriptions. In this section, we will show another aspect of our technique — application to method integration. The method integration plays an important role on constructing a new method from the fragments of existing methods[10, 13, 3]. In the specification development following Shlaer and Mellor's OOA, we should have four types of the diagrams — Entity Relationship Diagram, Object Communication Diagram, State Transition Diagram and Data Flow Diagram. They are meaningfully connected to each other to express a consistent specification. This meaningful connections can be formally specified in our framework as semantic constraints for the diagrams. In this modeling, the four diagrams can be considered as the basic fragments or parts of the methods for constructing another method Shlaer and Mellor's OOA, i.e. the method can be newly obtained as the result of the integration of the four existing methods.

Entity Relationship Diagram, Object Communication Diagram, and State Transition Diagram can be defined in Object-Z language in the same way as Data Flow Diagram. Figure 2 shows graphical representations, i.e. ER Diagrams of these diagrams, and it is useful to understand the following textual representations written in Object-Z notation, which are shown in the next page.



(a) Entity Relationship Diagram

(b) Object Communication Diagram

(c) State Transition Diagram

Fig. 2. ER Model Based Graphical Notation for Diagrams

EntityRelationshipDiagram

Product

$$\begin{array}{l}
entities : \mathbb{P}\, Entity \\
relationships : \mathbb{P}\, Relationship \\
entity\_attributes : \mathbb{P}\, Attribute \\
relationship\_attributes : \mathbb{P}\, Attribute \\
cardinalities : \mathbb{N} \times \mathbb{N} \\
domain\_of : Relationship ⤀ Entity \\
range\_of : Relationship ⤀ Entity \\
has1 : Attribute ⤀ Entity \\
has2 : Attribute ⤀ Relationship \\
has3 : Relationship ⤀ \mathbb{N} \times \mathbb{N} \\
\hline
status = \text{‘}completion\text{’} \\
\quad \Rightarrow WellFormedEntityRelationshipDiagram
\end{array}$$

$$\begin{array}{l}
WellFormedEntityRelationshipDiagram : \mathbb{B} \\
\hline
WellFormedEntityRelationshipDiagram = \\
\quad \operatorname{ran} domain\_of \subseteq entities \land \operatorname{ran} range\_of \subseteq entities \land \\
\quad \operatorname{dom} domain\_of = relationships \land \operatorname{dom} range\_of = relationships \land \\
\quad \operatorname{dom} has1 = entity\_attributes \land \operatorname{ran} has1 \subseteq entities \land \\
\quad \operatorname{dom} has2 = relationship\_attributes \land \operatorname{ran} has2 \subseteq relationship\_attributes \land \\
\quad \operatorname{dom} has3 = relationships
\end{array}$$

...

ObjectCommunicationDiagram

Product

$$\begin{array}{l}
objects : \mathbb{P}\, Object \\
messages : \mathbb{P}\, Event \\
send : Event ⤀ Object \\
receive : Event ⤀ Object \\
\hline
status = \text{‘}completion\text{’} \Rightarrow ConnectedGraph[objects, messages, send, receive]
\end{array}$$

...

StateTransitionDiagram

Product

$$\begin{array}{l}
states : \mathbb{P}\, State \\
transitions : \mathbb{P}\, Transition \\
events : \mathbb{P}\, Event \\
previous\_to : Transition ⤀ State \\
next\_to : Transition ⤀ State \\
input\_event : Transition ⤀ Event \\
output\_event : Transition ⤀ Event \\
\hline
status = \text{‘}completion\text{’} \\
\quad \Rightarrow ConnectedGraph[states, transitions, previous\_to, next\_to] \land \\
\qquad transitions = \operatorname{dom} input\_event
\end{array}$$

...

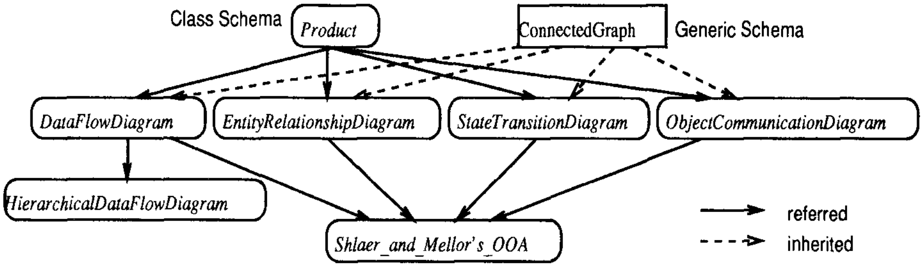


Fig. 3. Hierarchical Relationships among Schemas

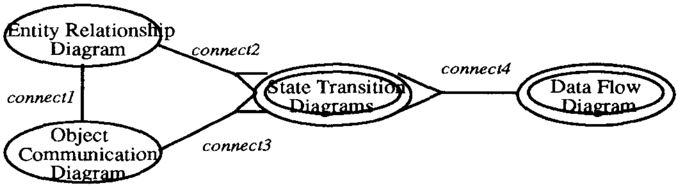

Entity Relationship Diagram
connect2
connect4
connect1
State Transition Diagrams
Data Flow Diagram
Object Communication Diagram
connect3

Fig. 4. Relation among Four Diagrams

To define these four diagrams, we have used the other schemas by schema reference and inheritance mechanisms. Figure 3 shows the hierarchical relationships among them. Method integration into Shlaer and Mellor's OOA will be done based on inheritance of these four diagrams.

Before defining Shlaer and Mellor's OOA, we will specify some constraints to integrate these four diagrams. Figure 4 shows the relationships among them. Connect1, connect2, connect3, and connect4 in the figure stand for mathematical constructs such as relation and functions which meaningfully connect the diagrams to each other. An entity relationship diagram is related to an object communication diagram which depicts the message flow among objects. An entity should occur as an object in the object communication diagram. For each entity occurring in an entity relationship diagram or each object in an object communication diagram, we have a state transition diagram which expresses its internal state change. Thus connect2 and connect3 have a set of functions which are from entities or objects to state transition diagrams.

Receive^{-1} occurring in the schema *connection_between_CD_and_STD* stands for the inverse map of the function *receive*. The formula in the predicate part of *connection_between_ERD_and_CD* specifies that the entities in Entity Relationship Diagram (*erd.entities*) are the same as the objects in Communication Diagram (*cd.objects*). The second formula in *connection_between_CD_and_STD* expresses that input messages to and output messages from an object should appear as input events and output events respectively in its state transition diagram.

connection_between_ERD_and_CD

$connect1 : EntityRelationshipDiagram \leftrightarrow ObjectCommunicationDiagram$

$\forall\, erd : \mathrm{dom}\, connect1;\ \forall\, cd : \mathrm{ran}\, connect1 \bullet erd.entities = cd.objects$

connection_between_ERD_and_STD

$connect2 : EntityRelationshipDiagram \leftrightarrow (Entity \twoheadrightarrow StateTransitionDiagram)$

$\forall\, erd : \mathrm{dom}\, connect2 \bullet erd.entities = \mathrm{dom}(\mathrm{ran}\, connect2)$

connection_between_CD_and_STD

$connect3 : CommunicationDiagram \leftrightarrow (Object \twoheadrightarrow StateTransitionDiagram)$

$$\begin{array}{l}
\forall\, cd : \mathrm{dom}\, connect3 \bullet cd.objects = \mathrm{dom}(\mathrm{ran}\, connect3) \\
\forall\, cd : \mathrm{dom}\, connect3;\ obj : \mathrm{dom}(\mathrm{ran}\, connect3);\ objtostd : \mathrm{ran}\, connect3 \\
\quad \bullet\ input\_event.objtostd(obj) = receive^{-1}(cd.objects) \\
\qquad \wedge\ output\_event.objtostd(obj) = send^{-1}(cd.objects)
\end{array}$$

connect_between_STD_and_DFD

$connect4 : StateTransitionDiagram \leftrightarrow (Event \twoheadrightarrow DataFlowDiagram)$

$\forall\, std : \mathrm{dom}\, connect4 \bullet std.output\_event \supseteq \mathrm{dom}(\mathrm{ran}\, connect4)$

Finally we can have the specification of OOA methods in the following :

Shlaer_and_Mellor's_OOA

DataFlowDiagram
CommunicationDiagram
EntityRelationshipDiagram
StateTransitionDiagram

connect_between_ERD_and_CD
connect_between_ERD_and_STD
connect_between_CD_and_STD
connect_between_STD_and_DFD

...

It consists of the specifications of the four diagrams and constraints for their integration, and holds the information about relationships among these diagrams in its state variables such as *connect1*, *connect2*, *connect3*, and *connect4*. Operations on *Shlaer_and_Mellor's_OOA* are inherited from the four diagram classes. For example, "IdentifyProcesses" of *DataFlowDiagram* is also an operation on *Shlaer_and_Mellor's_OOA*.

5 Conclusion

This paper has introduced another method modelling technique based on Object-Z to represent various methods for supporting specification development. It has been shown that our technique is applicable to method integration by using an example. The examples including method integration might be so simple that we could use a Z language instead of Object-Z. However, object-orientedness plays an important role on reuse of method fragments to construct new methods. Both Z and Object-Z are not executable, so we should combine our technique and other executable devices to enact the specified methods. The predicate logic underlying Z and Object-Z can provide the theoretical foundation for method integration and method synthesis. For example, we can check the consistency of the integrated method and the correctness of the integration process if the relevant methods are described in Object-Z. The logical formulas might be difficult for untrained persons to read and write. A library of method fragments, called *method base*[13, 11], are needed to specify and to integrate the methods.

References

1. A. Alderson. Meta-CASE Technology. In *Lecture Notes in Computer Science 509*, pages 81–91, 1992.
2. S. Brinkkemper. *Formalisation of Information Systems Modelling.* Thesis Publisher, 1990.
3. S. Brinkkemper. Integrating Diagrams in CASE Tools through Modelling Transparency. *Information and Software Technology*, 35(2):101–105, 1993.
4. M. Brough. Methods for CASE : a Generic Framework. In *Proc. of 4th International Conference CAiSE92, LNCS 593*, pages 525–545, 1992.
5. T. DeMarco. *Structured Analysis and System Specification.* Yourdon Press, 1978.
6. R. Duke, P. King, R. Rose, and G. Smith. The Object-Z Specification Language. Technical Report 91-1, Software Verification Center, University of Queensland, 1991.
7. A.K. Jordan and A.M. Davis. Requirements Engineering Metamodel : An Integrated View of Requirements. In *Proc. of 15th COMPSAC*, pages 472–478, 1991.
8. T. Katayama. A Hierarchical and Functional Software Process Description and its Enaction. In *Proc. of the 11th ICSE*, pages 343–352, 1989.
9. D.E. Knuth. Semantics of Context-free Languages. *Mathematical Systems Theory*, 2:127–145, 1968.
10. K. Kronlöf, editor. *Method Integration – Concepts and Case Studies.* Wiley, 1993.
11. M. Saeki, K. Iguchi, K. Wen-yin, and M. Shinohara. A Meta-Model for Representing Software Specification & Design Methods. In *Information System Development Process*, pages 149–166. North-Holland, 1993.
12. S. Shlaer and S.J. Mellor. An Object-Oriented Approach to Domain Analysis. *ACM SIGSOFT Software Engineering Notes*, 14(5):66–77, 1989.
13. K. Slooten and S. Brinkkemper. A Method Engineering Approach to Information Systems Development. In *Information System Development Process*, pages 167–186. North-Holland, 1993.
14. K. Smolander, K. Lyytinen, V.P. Tahvanainen, and P. Marttiin. MetaEdit — A Flexible Graphical Environment for Methodology Modelling. In *Proc. of 3rd International Conference CAiSE91, LNCS 498*, pages 168–193, 1991.
15. P. Sorenson, J. Tremblay, and A. McAllister. The Metaview System for Many Specification Environments. *IEEE Software*, 2(5):30–38, 1988.
16. J.M. Spivey. *The Z Notation — A Reference Manual.* Prentice Hall, 1987.