Derivation of Program Models for Web Application Systems
Using Meta-Models

Takakazu Kaneko, Nobuoki Mano
Graduate School of Informatics
Meisei University
2-590 Nagabuchi, Ome-shi, Tokyo-to 198-8655 Japan

Abstract: - We have been making researches on system design method for web application systems using models in knowledge base. Web systems are comprised of hetero-genius domains: user-interface, communication system, and database system. In this paper we propose a method using meta-models to derive Java source codes. This method, first, extracts program model skeletons by using knowledge about implementation patterns. Then, it derives concrete program models by using basic functional units. This method can be applied for problems over a wide range: from communication application in pure Java to communication using sessions in WEB system.

Key-Words: - Refinement, Model mapping, Meta-models, Program models, Planning

1 Introduction

We have been making researches on intelligent system design method for web application systems such as ATM system, schedule-management system, ticket-reservation system, and so on, based on logical approach using state-based specifications [1][2][3] expressed in a modified predicate logic. These web application systems are comprised of 3 hetero-genius domains: user-interaction, communication subsystem (client-server system), and database (at present, data structure handling program) [4]. We have presented the method of deriving upper-stream models from these uppermost-stream specifications [5]. We also proposed the refinement method of deriving program models from upper-stream models [6], where meta-models such as implementation patterns and Basic Functional Units (abbreviated as BFUs) for each domain mentioned above stored in our knowledge base are used. The pattern knowledge [7][8][9] bring us flexibility of system design.

In this paper we propose a mechanism for using meta-models to derive Java source codes. Implementation patterns and BFUs are domain-dependent general and meaningful units for constructing classes and methods and for constructing inside of method, respectively. This method, first, extracts program model skeletons consisting of class models and method models from implementation patterns. Then, it derives concrete program models including statements using BFUs.

As other works related with our method, MDA (Model Driven Architecture) [10] and Executable UML [11] are known. MDA is a method using UML (Unified Modeling Language) and OCL (Object Constraint Language) for leading platform-independent-model to platform dependent model. But it seems to fall into unnatural specifications because it uses specification descriptions based on class descriptions as its starting point. Executable UML is a tool for executing UML. It generates source codes by providing specification descriptions in a peculiar language independent from UML. But users cannot get into contact with its refining process, and cannot know how its transformation process is proceeding.

On the contrary, our method provides uppermost-stream specifications easy to understand for human designers, and refines them to program models based on knowledge. It also applies patterns to models at various stages in its refinement process so that we can observe how the refinement of target model proceeds. So we believe that our method is also useful for the education of software engineering.

The composition of this paper is as follows. Chapter 2 describes our software development process and its related models, especially focusing on models used in the middle-stream and down-stream of the process. Chapter 3 introduces implementation pattern knowledge for each domain mentioned above and explains how implementation patterns are used for deriving program model skeletons. Chapter 4 presents BFU knowledge and describes how these BFUs are used for deriving program model details. We use ATM system and Chat system (WEB type) as our examples throughout this paper.
2 Upper-stream models, program models, and meta-models of program models

2.1 Upper-stream models

Representation of the processing structure of a target system should have interaction command sequence and server-side database composition. Fig. 1 shows the classes and their associations of the ATM system. Not only creation of ConsortiumSlaveServer by ConsortiumMasterServer, but also creation of BankSlaveServer and UserAccountBase by BankMasterServer are shown with multiplicity information in the figure. The internal structure of PassDialog and that of database composition are shown in Fig. 7 and in Fig. 8, respectively.

2.2 Program models

The granularity of the ingredients of our program models are various, from coarse-grained to fine-grained. ClassM, FieldM, MethodM, ConstructorM, PartM and so on are the elements of our program model knowledge base. PartM corresponds to detail part in program model. Various aspect such as message send-receive operations, type conversion, basic operation, instance generation, type declaration, value assignment and so on can be represented in one model. PMPredicate, PMLogicalOperator, and so on are the constituents of these models. Program models are represented in tree structures using instances of these classes as their nodes. Fig. 2 shows model description for the sending operator using ObjectStream. The left hand side of the symbol ‘;’ in argument parts represent input arguments, and the right hand-side represents output arguments, which does not exist in this example. Strings headed with ‘?’ are variables.

2.3 Meta-models

Patterns contain meta-model parametric representation to be specialized to fit with target problem. This enables flexible design development. They consist of precondition, post-condition, and meta-code. Condition expressions are represented by modified first-order predicate logical formulae. Meta model description forms are same with model description forms, except for the use of role expressions. Fig. 3 shows meta-model of the model in Fig. 2. Role expressions (‘| role name’) are used for those parts that are changeable depending on target problems.

2.3.1 Implementation patterns

An Implementation pattern has definitions of association in each component (including multiplicity between relating components) and meta-model description (including meta-code corresponding to component). Definition of associations describes in “Impassdef (which is an abbreviated string of IMPlementation ASSociation DEFinition) - end” and each meta-model description describes in “meta-model descriptions - end”. Mapping of role and value are decided by pattern matching.

For example, fig. 4 shows a part of multi-client-server description. The right part and the left part of symbol @ represent operation name (e.g., send_account) in model description forms are same with model description in Fig. 2 and Java code, respectively. Concrete codes are given in place of symbols <> , where server and client can be replaced with Consortium and ATMClient when Consortium is a Server and ATMClient is a Client. Source codes are obtained by substituting assigned values for corresponding role expressions.

2.3.2 BFUs

BFU description has definitions of association (including multiplicity between relating classes) and meta-model description (including meta-code corresponding to class). Meta-model description has various types such as preparation, communication and so on. Definition of associations describes in “Bfuassdef (BFU ASSociation DEF)-end” and each meta-model description describes in “communication -end”.

Fig. 1 Class-Association Diagram of the ATM System.

Fig. 2 Description of account sending part in ATM system.

Fig. 3 Meta-program-node description in sending operator for the ObjectStream.
Socket BFU and ObjectStream BFU for communication domain, JFrame BFU using dialogbox for user input for GUI domain, and HashMap BFU as the implementation of functional dependency relation for database are representative BFUs. Descriptions of them are given in Fig.5.

2.3.3 Auxiliary chips

Auxiliary chips such as group of knowledge on type conversion and primitive atomic operations are indispensable units of meta-models for our system. Group of knowledge on type conversion (e.g., Integer.parseInt()) are used when data-type conversion are required to fill the gaps between postconditions and preconditions of instantiated BFU pairs. Group of atomic operation (for example, introduction of variables to generate assignment statements) are required when enabling plural reference to object instances and when extracting values of field variables of objects.

2.4 Example problems used for program model generation

Our program model generation process consists of two phases: phase of deriving program model skeletons using implementation patterns (Section3.2) and phase of generating detailed program models using BFU (Section4.2). We use two example problems. Programs of ATM system is an ordinary Java application, and uses the Java awt for GUI. Programs of chat system is a Web-program based on Tomcat [12], consisting of the interaction phases: Enter-Password for account certification, EnterRoom and ExitRoom for entering and exiting from the chat room, InputMSG for inputting messages, and so on.

In our system program models are represented in graphical models, but to make readers easy to understand the construction process, we show Java program codes corresponding to (or, in another words, codes derived from) the program model in the following part of this paper.

3 Design decision and related handling of models

3.1 Design decision in the refinement process

First one of architecture patterns is selected to determine the communication method in the target system. The ordinary communication method in Java applications (single client-server or multi-client server) or the web communication using Tomcat.

Then we make trials to map from elements and relationships in meta-models of program models to those in upper-stream models based on design decision. As the result of successful mapping, we can obtain main elements of program models for each domains.

We can also obtain state assertions of system states in program models of target systems from state assertions of system states in upper-stream models using those mapping relationships.
3.2 Use of patterns

Fig. 6 shows the way to use patterns. The matching of an implementation pattern with problem processing structure in uppermost stream models derives a program model skeleton and class association diagram. Then, the matching of BFU with the derived class association diagram derives detailed program model. The derivation details are described in section 4.2.

Our refinement method tries to map meta-models of program model to upper-stream models, taking multiplicity constraints into consideration. When successful mappings are possible, program models are obtained, replacing parameters of meta-models by their corresponding part of upper-stream models as their values.

The left figure of Fig.7 shows the mapping from dialogbox with two instances belonging to TextField class (program meta-model) to upper-stream model (dialog with user account number and his password), and the right figure shows an instance of DialogBox consisting of two instances of TextField with the names accountNum and password (the obtained part of program model as the result of the mapping).

Fig.8 shows the mapping from HashMap meta-model to the database part in the upper-stream model. The latter has, as its columns, Account, Name, Password, and Balance. Column Account is the key and Columns Name, Password, and Balance are functionally dependent on the column Account. As the result of mapping, HashMap is instantiated to the one in program model with Account as its key and <Name, Password, Balance>, integrated into the fields of Person class, as its value.

3.3 Derivation of program model skeletons by the application of implementation patterns

3.3.1 Example 1 – ATM system

By the application of implementation patterns for upper stream model of ATM system, we can obtain program model skeleton corresponding to Java codes shown in Fig.9. We explain the detail of the figure.

represents the parts through the introduction of dialog boxes. PassDialog, OperationDialog and BalanceDialog are derived from EnterAccount, Withdraw and Deposit interaction-phase, respectively. Because everything is built in the pattern beforehand, common part information such as upper class of a class, interface, exception handling, assignment of initial value, statement of Dialog display and so on appear as it is as a code. ConnectInfo class and its related interaction codes are introduced to obtain dialog boxes.

depicts the parts by the use of HashMap meta-structure. Deposit method and calling the method are derived from the deposit interaction-phase. 

depicts a part of code generated by applying client-server pattern/architecture pattern. BankMasterServer and BankSlaveServer are divided from Bank based on the processing structure, and are understood that BankMasterServer repeatedly generates the instance of BankSlaveServer from information on the multiplicity. So, the while statement that generates the instances is introduced as for Java code. <1>, <2>, <3> in the figure indicate the area for program model details to be inserted (shown as detailed codes in Fig.11) in the next section.

3.3.2 Example 2 – Chat system

In the case of applying the tomcat pattern, a class(enterPassword, enterRoom, InputMSG, exitRoom and so on) is assigned to each interaction-phase.
4 BFUs and their use for deriving program model details

When the construction-phase of program model skeleton has completed, our system moves into the derivation phase of details of program model.

4.1 Model fusion by unification

Program models produced by implemented patterns and instantiation of BFU meta-models of each domain have to be merged into consistent models which satisfy assertions of refined system states or input-output assertions of introduced functions. Unification between refined system state assertions, pre-conditions of operators, and post-conditions of operators with some patching of instantiated auxiliary chips bring us consistent merged program models including data-flow.

Fig.10 explains the fusion of elements of instantiated models in different domains introduced by refinement. By the unification of the post-condition of input-Account operator with the pre-condition of send_enterAccount operator, the output variable representing account input information transferred from the input-Account operator proves to be passed to the input variable of send_enterAccount operator. The dataflow formation makes account information in Client component to be sent to its corresponding Server component.

4.2 Derivation of program model details

The selection and application of BFUs are done in the following order - structure matching, parts generation, plan synthesis[13], and program tree update. Matching of multiplicity constraints must be satisfied in structure matching.

4.2.1 Example 1 – ATM system

Detailed program model for the ATM system are obtained as the result of applying various BFUs. Fig.11 shows the Java codes corresponding to the program model.

(a) Socket BFU

First, communication type BFU such as socket selects communication classes. Then, user appropriately decides information necessary for establishing communication (by introducing the other BFU to be resolved, or, by obtaining information from introduced parts). These necessary outside information are built into BFUs beforehand. For ATM example, connection of ATMClient and ConsortiumMasterServer is established by applying Socket BFU, where the required server name and port number are provided from some source (from argument of main method in this example). Fig.11 focuses on client side.

In this case, the statement that performs appropriate processing is also introduced based on prepared mapping of natural type and role type (by the use of Integer.parseInt()).

(b) ObjectStream BFU

ObjectStream BFU also belongs to communication type BFUs. Fig.3 shows this BFU description. In each role expression, when Socket BFU is applied, |ostream| has |socketOut| and |istream| has |socketIn|. |istream| has |data| according to the sending data such as |?pass|, |?socketOut| and so on of the action in the interaction-phase. |?socketIn| shows instance of ObjectOutputStream. This is converted into |oos| in Fig11 because the value is memorized in oos. It is similar for |?socketOut|. |?socketIn| is converted in transferData1.account by the data-flow information.

Generation of ObjectOutputStream instance requires OutputStream from a Socket. If Socket BFU has not been applied before this stage, the application of
ObjectStream BFU is interrupted, and Socket BFU is applied.

(c) awt Button BFU
GUI type BFU with its structure description matches the GUI class with its structure in target systems.

In ATM example, matching is performed with the class PassDialog in ATMClient. The code in the <2> column in Fig.11 shows setting up a button in PassDialog. In addition, actionPerformed method and common handling is added to PassDialog to process event from the button.

(d) basic data processing BFU
Data structure BFUs such as HashMap BFU (in Fig.5) are often used for representing database such as AccountBase in ATM system (in Fig.8). BankSlaveServer performs transaction operation and process AccountBase. So person information is extracted and stored as shown in the codes <3-1>, with the result of addition of balance <3-2> in Fig.11.

4.2.2 Example2 -- Chat system
Fig.12 shows Java code obtained for the chat system:
(a') the codes by the application of the Session pattern.
(b') the codes by the application of the Servlet pattern.

WEB type target system requires information saving operations for each session, because information is lost at each disconnection. It also requires the Servlet pattern to be used for reflecting the transition between pages in the code. The transition-ahead can be reasoned based on its class association diagram.

These parts set are synthesized to a consistent plan and program-tree by our incremental planner.

5 Conclusion
We proposed a derivation method of Java source codes using meta-models, namely implementation patterns and basic functional units (BFUs), from uppermost stream specifications and upper stream models. We explained the derivation process using the ATM simulator and Chat System as examples. Refinement method using meta-models provides flexible design development for us. We have been engaged in the task of implementing our method to a real software development system.

References: