Refactoring-Based Stepwise Refinement in Abstract System-Level Design

Ryosuke Yamasaki[1], Kazutaka Kobayashi[1], Nurul Azma Zakaria[1], Shuji Narazaki[2], and Norihiko Yoshida[1]

[1] Saitama University, 255 Shimo-Ohkubo, Sakura-ku, Saitama, Japan
{ryosuke, kazutaka, azma, yoshida}@ss.ics.saitama-u.ac.jp
[2] Nagasaki University, Bunkyo 1-14, Nagasaki, Japan
narazaki@cs.cis.nagasaki-u.ac.jp

Abstract. Stepwise refinement in system-level design corresponds to restructuring an internal structure of a system while preserving functions of the system. We are aiming to build the restructuring process based on refactoring techniques. In this paper, we describe a restructuring procedure to obtain a concrete specification description from an abstract one. Moreover, we describe some existing refactorings used in restructuring steps and a new refactoring for system-level design. We designed a simple internet-router as an example. As a result, we obtained a specification model defined in the SpecC methodology from an abstract one. Moreover, our proposal shows that our research opens a new application field of refactoring, refactoring can be applied sufficiently to system-level design, and the refactoring can be the basis of stepwise refinement.

1 Introduction

System-level design methodology, which is aiming to derive interactively and cooperatively a cycle-accurate hardware implementation and software implementation from an abstract specification where hardware and software are not distinguished and concept of time is abstracted, obtains remarkable results to improve design productivity of system-LSI/System-on-Chip. Some programming languages, called system-level description languages, are proposed in order to realize the design methodology. SystemC [1] and SpecC [2] have a concept of object-orientation. As a methodology, the SpecC design methodology [2] based on SpecC is well-defined and systematized.

Recently, not only languages but also design methodologies based on object-orientation are being researched actively. When designers represent the first system specification in object-oriented modeling languages such as UML, designers must satisfy some description constraints of models defined in a methodology.

For example, a specification model in the SpecC methodology has some description constraints; such as (1) a series of related functions are grouped as a behavior; (2) channels, ports, and global variables are used to connect behaviors; and (3) behaviors access to own ports to communicate each other. However,

E. Sha et al. (Eds.): EUC 2006, LNCS 4096, pp. 712–721, 2006.

in a modeling method in object-oriented design, a function is grouped by data and procedures related to the data. Moreover, to separate a computation and a communication by (2) and (3) is enabled after (1).

There are two ways to bridge the gaps between a specification model and a model of object-oriented design; (a) representing a specification model in UML directly, or (b) transforming a system specification in UML to a specification model. In case of (a), designers must consider a structure of functions when describing the first specification. In case of (b), it is necessary to establish a concrete transformation procedure preserving the functions of the system.

Restructuring method of a system structure while preserving its functions have been researched in program semantics and software engineering. Program transformation and refactoring have been established.

We are aiming to systematize stepwise refinement steps based on refactoring in order to realize (b). Preliminary result is shown in [3], it focused on obtaining a concrete system description in SpecC from an abstract one in Java. In this paper, we focus on the earlier phase than a creation phase of a specification model in SpecC methodology. Moreover, we describe concrete procedures based on refactoring in order to derive the specification model in SpecC from an executable specification that is created from a system specification in UML.

The remainder of this paper is organized as follows. In section 2, we describe system restructuring techniques, particularly the refactoring. In section 3, we describe our restructure procedures and a new refactoring for system-level design. In section 4, we describe an example design and its result. In section 5, we introduce related works, and section 6 describes the conclusions.

2 System Restructuring Techniques

There are two methods to restructure a system description to another one by rewriting the internal structure of the system, while preserving the external behavior/functions of the system, program transformation and refactoring. However, compilation and optimization in compiling are not included in the two restructuring methods because the description form is preserved.

2.1 Program Transformation

Program transformation is based on logic and mathematic, and it is a theoretical method that rewrites a program following transformation rules whose validity are guaranteed strictly. Therefore, this method enables to restructure an internal structure of a system preserving strictly its external behavior. Partial evaluation and meta-programming might be included. This method is studied for program customization, derivation, efficiency improvement, and so on. However, there are few application experiences to a large/complex program because transformation rules, procedures, and system specification are described as logical/mathematical expressions.

2.2 Refactoring

High reusability is one of the most important advantages of object-oriented design. However, it is actually too difficult to draw it to its maximum from the first design. Then, refactoring is systematized as a method that improves maintainability, readability, reusability, and modularity in later design phase by restructuring a system.

Refactoring is a collection of program rewriting rules for each situation and purpose. The collection is systematized on the purposes of rewriting and program structures. For example, *Extract Class*, *Form Template Method*, and *Move Method* are refactoring names, and a program code of *Extract Method* in Java is shown in Fig. 1. *Extract Method* is used to reduce repeat codes by aggregating fragments of the code, and to improve modularity and readability by decomposing too longer method. In Fig. 1(a), console output statements in `for` block (line 4–5) are extracted as a method `multiDisp` shown in Fig. 1(b), line 4–7. As a result, readability of the method `display`, reusability of the method `multiDisp`, and modularity of the whole system are improved. Details of other rewriting rules cited later by its name are shown in reference [4].

Refactoring is a practical method, and a theoretical strictness is not guaranteed completely. Instead, it is easier than program transformation to apply rewriting-rules to restructure a real-world system. For the above-mentioned reasons, we adopt refactoring.

```
1:void display(int t){
2: int i;
3: printHeader();
4: for(i=0;i<t;i++){
5:  System.out.print(i);}}
```

(a) Original.

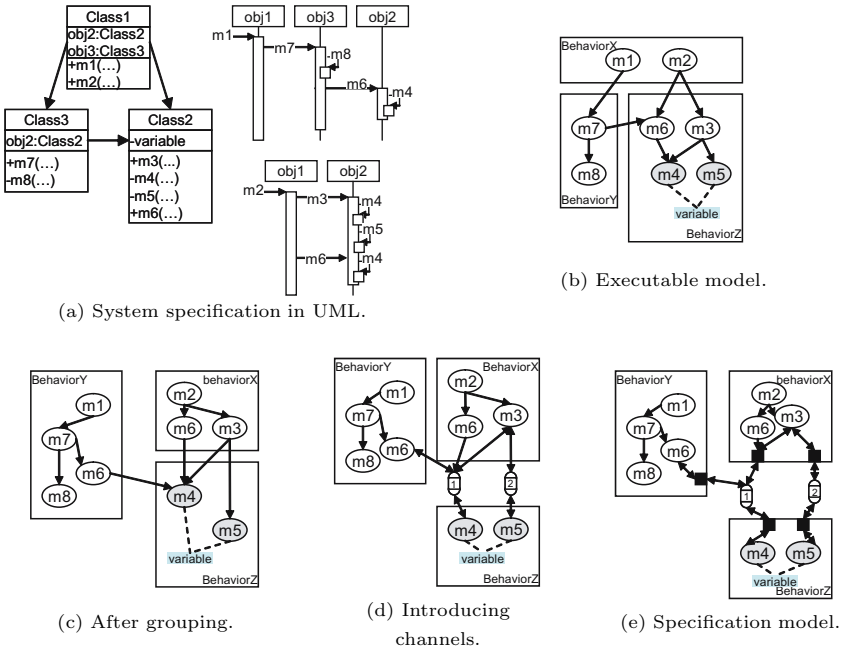
```
1:void display(int t){
2: printHeader();
3: multiDisp(t);}
4:void multiDisp(int t){
5: int i;
6: for(i=0;i<t;i++){
7:  System.out.print(i);}}
```

(b) Method extracted.

Fig. 1. Example: Extract method

3 Outline of Proposed Methodology

A restructuring of system descriptions consists of four steps; (1) representing a system specification, (2) generating an executable specification from (1), (3) re-grouping, and (4) replacing behavior-method call to channel-method call and adding channel-port. Fig. 2 shows changes of a system structure in proposal method. In Fig. 2(c), it assumes that a grouping policy is shared-memory communication. In Fig. 2, a rectangle is an object or a behavior, an oval is a method, an arrow is a method call, a broken line is an access to a variable, and a black rectangle is a port.



(a) System specification in UML.

(b) Executable model.

(c) After grouping.

(d) Introducing channels.

(e) Specification model.

Fig. 2. Outline of design flow

3.1 Restructuring Flow

1. Specification Representation. Designers represent a system specification in UML (Fig. 2(a)). At this abstract level, computation and communication are not separated. Following the concept of object-oriented design, a grouping unit is the collection of the data and the procedures related to the data.

2. Executable Specification Description. Designers describe an executable specification in SpecC by relating an object and a behavior. All methods that should be public are defined in `interface` referring class diagrams or sequence diagrams. For example, the `BehaviorZ` implements `m3` and `m6`. All behaviors can communicate each other by method call. We call such public method "behavior-method". Behavior-methods are defined in `interface`, and we call such interface "behavior-interface". For example, in Fig. 3 shown later, the line 1–3 are behavior-interface, and line 5–7 are behavior-method. The behavior `Master` can access to behavior-method directly (line 14 in Fig. 3).

At this abstraction level, designers verify through simulation and test that required functions are satisfied. After that, to simplify a latter restructuring, all behaviors are applied *Self Encapsulate Field*. By this refactoring, all methods that access to internal variables are limited to accessor (getter/setter method). In Fig. 2(b), the method `m4` and `m5` are accessors.

3. Re-grouping. Related functions are grouped by each start method of a method call chain. A method called by other methods can belong to all behaviors

```
 1:interface SlaveIF{                          //behavior-interface
 2:  int getArea(int height, int width);
 3:};
 4:behavior Slave() implements SlaveIF{
 5:  int getArea(int height, int width){    //behavior-method
 6:    return height * width;
 7:  }
 8:  void main(void){}
 9:};
10:behavior Master() {
11:  int height = 3, width = 5, area = 0;
12:  Slave slave();
13:  void main(void){
14:    area = slave.getArea(height, width); // behavior-method call
15:  }
16:};
```

Fig. 3. Before replace

where a caller method belongs to, or can belong to a new created behavior. For example in Fig. 2(b), the method `m1`, `m6`, `m7` and `m8` belong to `BehaviorY`, and the method `m2`, `m3` and `m6` belong to `BehaviorX`. First, by applying *Move Method* to those methods, they are moved to `BehaviorX` and/or `BehavorY`. Next, behavior-interfaces are re-defined according to changes after applying *Move Method*. Finally, three groups, `BehaviorX`, `BehaviorY` and `BehaviorZ` are created.

4. Introduce Channel and Add Channel-Port. Designers must replace behavior-method call to channel-method call. "Channel-method" becomes public by implementing interfaces, and we call a calling to it "channel-method call".

First, designers classify behavior-method calls. For example in Fig. 2(c), they are classified into two, `m4` and `m5`. Second, designers check up sent/received data by the method call, and define channel according to the data. Third, designers rewrite a code calling a behavior-method to a channel-method call. After that, designers remove an `implements` code. As a result, designers obtain an executable specification shown in Fig. 2(d). Continuously, designers add ports to behaviors, and rewrite a channel-method call to a channel-port access.

From the above restructuring, designers obtain a rewritten description shown in Fig. 2(e). Channel definition, rewriting to channel-method call, and adding channel-port are described in 3.2.

3.2 Replace Behavior-Method Call to Channel-Method Call

Here, we describe a new refactoring for system-level design, *Replace Behavior-method call to Channel-method call*. In a specification model in SpecC methodology, behaviors communicate data by using channels, ports, and global variables. Moreover, it is necessary to specify the input/output of data.

To satisfy those description constraints, first, channels are defined according to parameters, arguments and returned value. Next, designers replace behavior-method call to channel-method call. The replacing procedures are described below with Fig. 3, Fig. 4 and program code.

Designers subdivide a behavior-method call into four methods, send argument, receive argument, send result, and receive result (line 3–6 in Fig. 4 respectively).

```
 1:#define INVALID -99999
 2:interface GetAreaIF{                                  //channel-interface
 3:  void reqSend(int height, int width);
 4:  void reqRecv(int *height, int *width);
 5:  void ackSend(int area);
 6:  void ackRecv(int *area);};
 7:channel GetAreaCh() implements GetAreaIF{
 8:  int height=INVALID, width=INVALID, area=INVALID;
 9:  event req1, req2, ack1, ack2;
10:  bool lock=false; event release;                     // for concurrent access control
11:  void reqSend(int arg1, int arg2){                   //channel-method
12:    while(lock){wait(release);} lock=true;           // for concurrent access control
13:    height=arg1; width=arg2; notify(req1); wait(req2);}
14:  void reqRecv(int *arg1, int *arg2){                //channel-method
15:    while(height == INVALID){wait(req1);}
16:    (*arg1)=height; (*arg2)=width;
17:    notify(req2); height=INVALID; width=INVALID;}
18:  void ackSend(int arg){                             //channel-method
19:    area=arg; notify(ack1); wait(ack2);}
20:  void ackRecv(int *arg){                            //channel-method
21:    while(area==INVALID){wait(ack1);}
22:    (*arg)=area; notify(ack2); area=INVALID;
23:    lock=false; notify(release);}};                  //for concurrent access control
24:behavior Slave(GetAreaIF getAreaCh){
25:  int getArea(int height, int width){return height * width;}
26:  void main(void){
27:    int height = 0, width = 0, area = 0;
28:    getAreaCh.reqRecv(&height, &width);              //receive argument
29:    area = getArea(height, width);                   //original processing
30:    getAreaCh.ackSend(area);}};                      //send result
31:behavior Master(GetAreaIF getAreaCh) {
32:  int height=3, width=5, area=0;
33:  void main(void){
34:    getAreaCh.reqSend(height, width);                //send argument
35:    getAreaCh.ackRecv(&area);}};                     //receive result
```

Fig. 4. After replace

Second, a new channel implementing them is defined (line 7–23 in Fig. 4). Third, designers add a channel-port by applying *Add Parameter* to a calling/called behavior (line 24 and 31 in Fig. 4). A code of behavior-method call in the calling behavior (line 14 in Fig. 3) is rewritten to "send argument" and "receive result" (line 34 and 35 in Fig. 4 respectively). Also, codes of "receive arguments", "original processing", and "send result" are added to `main` method (line 28–30 in Fig. 4 respectively). In case of Fig. 4, to realize exclusive access, designers insert line 10, 12, and 23 in Fig. 4 to channels.

4 Example Design and Result

As an example design, we adopted internet-router. We designed three kinds of router, "distance vector", "link state", and "path vector". In addition, the specification is limited to only basic functions because the purpose of this experiment is not designing routers. In this paper, we explain the distance-vector router.

We used UML to represent the first specification with Pattern Weaver 1.0 [5], and SpecC languages to describe executable specifications with VisualSpec 3.5 [6]. To simplify, we prohibited dynamic instantiation, instance transference, and recursion call.

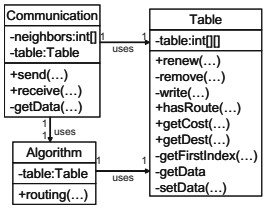


Fig. 5. System specification in a class diagram

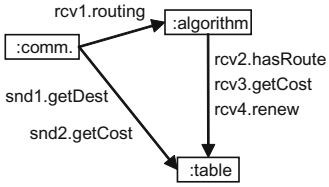


Fig. 6. System specification in a communication diagram

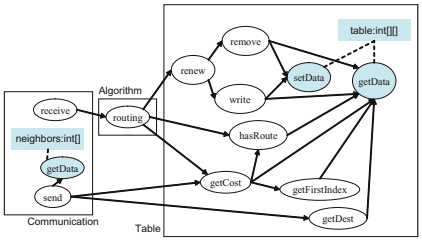


Fig. 7. Executable model

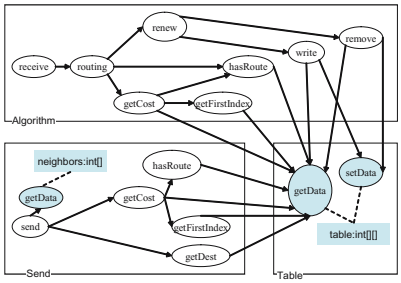


Fig. 8. After grouping

4.1 Design Flow

1. Behavior of Router. From use case analysis, we obtained three basic functions, exchanging route information, route exploration, and management of a route table. Moreover, we defined three classes, Communication, Algorithm, and Table. We described not only class diagram, usecase description, and usecase diagram but also sequence diagram, communication diagram, and so on. In this paper, only class diagram and communication diagram are illustrated in Fig. 5 and 6 respectively.

The router behaves according to the below procedures, `rcv1`, `rcv2`, `rcv3` and `rcv4` in Fig. 6, when the router receives a route information. And the router behaves according to the below procedures, `snd1` and `snd2` in Fig. 6, when the router sends own route information to neighbors.

2. Executable Specification Description. We described an executable specification in SpecC according to the previous specification. By relating a class to a behavior, we defined three behaviors, "Communication", "Algorithm", and "Table". The executable specification is illustrated in Fig. 7. Oval, rectangle, arrow, and a broken line represent method, behavior, method-call, and an access to variables respectively.

3. Re-grouping. Each start method, `receive` and `send`, and other methods relating to the start method belong to two behaviors. These two behaviors,

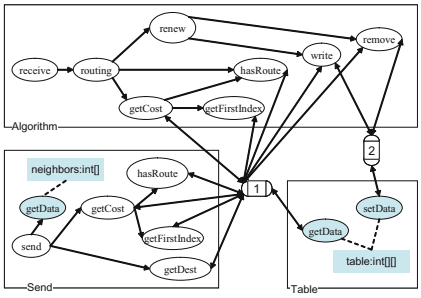


Fig. 9. Introducing channels

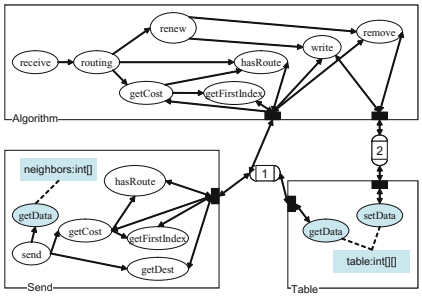


Fig. 10. Specification model

`Rreceive` and `Send`, are created by dividing `Communication`. The internal variable `table` in `Table` and accessors are left in `Table`. In addition, the internal variable `neighbors` in `Communication` belongs to `Send`.

The behavior `Receive` is created by applying *Extract Class* to the method `receive` in `Communication`. After applying, only `send` and accessor are left in `Communication`. To rename `Communication` to `Send`, *Rename* is applied to the behavior `Communication` because the main function of `Communication` is only to send route information. There are eight methods belong to the behavior `Receive` and six methods belong to the behavior `Send`. To move those methods to each behavior, *Move Method* is applied to each method. To rename `Receive` to `Algorithm`, *Rename* is applied to the behavior `Receive` because the main function of `Receive` is execution of routing algorithm.

As a result of the above restructuring, we obtained a re-grouped description shown in Fig. 8.

4. Introduce Channel and Add Channel-Port. In Fig. 8, there are two behavior-method calls, `getData` and `setData`. Therefore, we defined eight channel-methods and two channels according to the behavior-method by applying 3.2. From these restructuring, we obtained the executable specification shown in Fig. 9. Next, we added channel-ports to behaviors `Algorithm`, `Send`, and `Table` by applying 3.2. From these restructuring, we obtained an executable specification which can satisfy description constraints of the specification (Fig. 10).

4.2 Experimental Results and Discussions

It was confirmed that the specification model (Fig. 10) behaved just same as the first executable specification (Fig. 7) by the simulation. Moreover, the bug was not mixed by rewriting at all, and there was not returning to previous steps in this experiment either.

The policy of grouping the relating function is various. For example, the method `getCost` and other related methods can be grouped together as a behavior `GetCost` illustrated in Fig. 11 and Fig. 12. As a result, the behavior `Algorithm` must wait until the behavior `GetCost` is released, and vice versa.

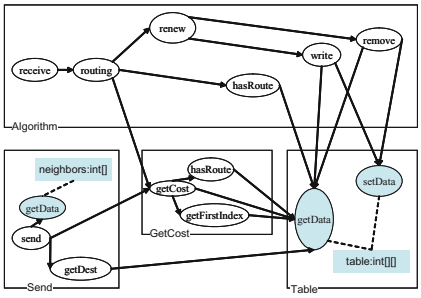


Fig. 11. Executable model, sharing the behavior `GetCost`

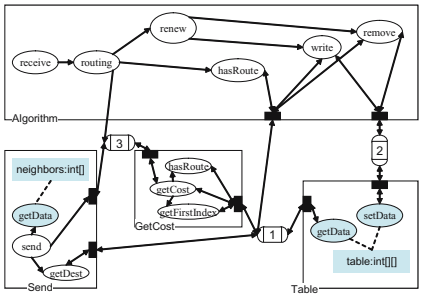


Fig. 12. Specification model, sharing the behavior `GetCost`

Designers can restructure a system description to various structures by using refactoring, while considering the trade-off such as processing time.

5 Related Works

[7] and [8] use to describe a system specification in UML and SystemC. [7] enables to generate SystemC codes from a system specification in UML using UML-profile. However, designers need to consider the structure and connection relation of function modules when describing a system specification. [8] also generates SystemC codes from a system specification in UML using Rational Rose RT tool. In this research, some descriptions in UML are limited to represent connection and communication between modules. These two researches belong to the approach (a) explained in section 1.

There are some researches applying MDA [9] to SoC design such as [10] and [11]. However, it is difficult to confirm through simulation whether behaviors of a system are correct or not. In our method, designers can trace and control any refinement steps, and any anytime the specification is executable.

Program derivation of a systolic array from a mathematical specification [12] is an application of program transformation to system-level design. This brings the completely validated implementation formally because based on program transformation. However, the application example to the real-world design is few because same reason. In our method, it is easier than program transformation to apply to a real-world design because our method based on refactoring.

6 Summary and Conclusions

In this paper, we described an application of existing refactoring to system-level design, and proposed a new refactoring for system-level design. Additionally, we described concrete refinement steps that refine an executable specification based on refactoring. Designers can restructure a system specification in safely, because

the external behaviors (or functions) are preserved. Proposal restructuring steps enable designers to describe a system specification at higher level.

The most important of future works is an automation of whole restructuring flow. Some simple refactoring are automated in some integrated development environments (IDE). We aim to realize a design automation based on our proposal, referring those tools.

References

1. T. Grötker, S. Liao, G. Martin, and S. Swan, "System Design with SystemC", Kluwer Academic Publishers, 2002.
2. Daniel D.Gajski, J. Zhu, R. Dömer, A. Gerstlauer, and S. Zhao, "SpecC: Specification Language and Methodology", Kluwer Academic Publishers, 2000.
3. R. Yamasaki, K. Kobayashi, N. Yoshida, and S. Narazaki, "Application of Refactoring Techniques to Abstract System Level Design", IEICE/IPSJ Information Technology Letters, Vol.4, September 2005.
4. M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts. "Refactoring", Addison-Wesley, 1999.
5. Pattern Weaver 1.0, Technologic Arts Inc., url: http://pw.tech-arts.co.jp/pw/index.html
6. VisualSpec 3.5, InterDesign Technologies Inc., url: http://www.interdesigntech.co.jp/english/
7. E. Riccobene, P. Scandurra and A. Rosti, "A SoC Design Methodology Involving a UML 2.0 Profile for SystemC", Proc. of the Design, Automation and Test in Europe (DATE'05), Vol. 2, pp.704–709, March, 2005.
8. W.H. Tan, P.S. Thiaggarajan, W.F. Wong, Y. Zhu and S.K. Pialakkat, "Synthesizable SystemC Code from UML Models", 41th Design Automation Conference (workshop UML for SoC Design), 2004.
9. MDA, url: http://www.omg.org/mda/
10. P. Boulet, J.-L. Dekeyser, C. Dumoulin, and P. Marquent, "MDA for SoC Embedded Systems Design, Intensive Signal Processing Experiment", SIVOES-MDA workshop at UML2003, pp.20–24, October, 2003.
11. Stephen J. Mellor, John R. Wolfe, Campbell McCausland, "Why System-on-Chip Needs More UML like a Hole in the Head", Proc. of the Design, Automation, and Test in Europe (DATE'05), Vol. 2, pp.834–835, 2005.
12. N. Yoshida, "Transformational Derivation of Highly Parallel Programs", Proc. 3rd Int'l Conf. Supercomputing, Vol.3, pp.445–454, Boston, 1988.