Sketch-Based Gradual Model-Driven Development

Peiyuan Li Naoyasu Ubayashi Di Ai Yu Ning Li
Shintaro Hosoai Yasutaka Kamei
Kyushu University
Fukuoka, Japan
{lipeiyuan@posl.ait, ubayashi@ait, aidi@posl.ait, liyuning@posl.ait, hosoai@qito, kamei@ait}.kyushu-u.ac.jp

ABSTRACT

This paper proposes an abstraction-aware reverse engineering method in which a developer just makes a mark on an important code region as if he or she draws a quick sketch on the program list. A support tool called *iArch* slices a program from marked program points and generates an abstract design model faithful to the intention of the developer. The developer can modify the design model and re-generate the code again while preserving the abstraction level and the traceability. *Archface*, an interface mechanism between design and code, plays an important role in abstraction-aware traceability check. If the developer wants to obtain a more concrete design model from the code, he or she only has to make additional marks on the program list. We can gradually transition to model-driven development style.

Categories and Subject Descriptors

D.2.11 [**Software Engineering**]: Software Architectures—*Information hiding*

General Terms

Design; Languages

Keywords

MDD; Interface; Abstraction; Traceability

1. INTRODUCTION

MDD (Model-Driven Development) is one of the promising approaches to software abstraction. An application can be developed at a high abstraction level by using a DSL (Domain-Specific Language), a DSML (Domain-Specific Modeling Language), or an ADL (Architectural Description Language). We do not need program code, because it can be fully generated from its design model.

However, it is not easy to migrate from a traditional development style to MDD, because a large quantity of legacy programs already exists in industry. Moreover, design documents and programs are not necessarily consistent. Major MDD tools adopt a code generation approach in which we have to create a design model and then translate it to the corresponding code. However, it is unrealistic to abandon existing code and migrate to MDD by restructuring design models from scratch. Although reverse engineering tools can reproduce a design model from the existing program code, the model tends to be meaningless because its abstraction level is the same to that of the original code. A design model should represent an software architecture and be more abstract than the code. Of course, the role of design is not limited to abstraction. For example, a developer has to take into account non-functional requirements. However, abstraction is critical in software development [9].

To deal with this problem, we propose an abstraction-aware reverse engineering method in which a developer just makes a mark on an important code region as if he or she draws a quick sketch on the program list as illustrated in Figure 1. A support tool called *iArch* [1] slices a program from marked program points and generates an abstract de-

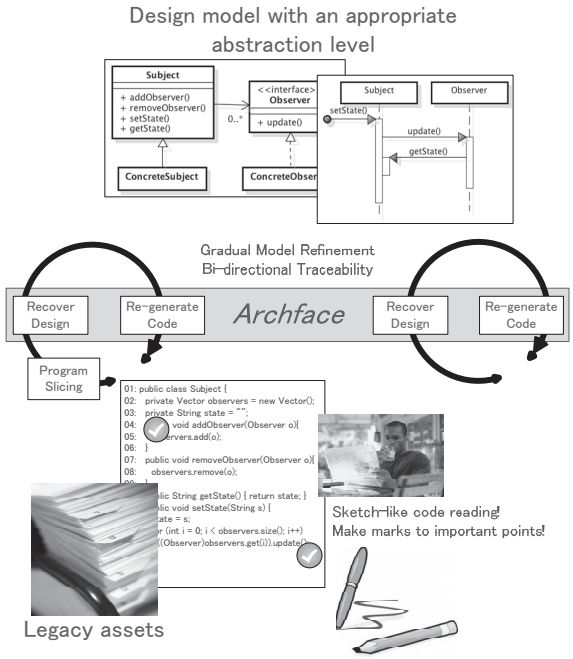

Figure 1: Gradual MDD

InnoSWDev'14, November 16, 2014, Hong Kong, China
Copyright 2014 ACM 978-1-4503-3226-2/14/11...$15.00
http://dx.doi.org/10.1145/2666581.2666595

sign model faithful to the intention of the developer. This mark is not directly embedded in the code but just an input to the *iArch*. Special annotations that invade separation on concerns are not needed in the code. The abstraction level of a generated model is determined by specified marks. An interface mechanism termed *Archface* [14, 15] is used to preserve the traceability between a generated design model and the original code. We can modify the model and re-generate the code again while preserving the abstraction level and the traceability. If a developer wants to obtain a more concrete design model from the code, he or she only has to make additional marks on the program list. These marks indicate extra design concerns that are reflected to the design model. For example, the developer only has to make a new mark on a method definition if the developer is aware that the method is important and should be contained in the design model after reading the code in more detail. We can gradually transition to the MDD style.

Our methods are summarized as follows: 1) an abstract design model can be automatically generated by sketch-like program reading; 2) traceability between design and code is always maintained using *Archface*; and 3) a design model can be gradually refined whenever we obtain new knowledge from the legacy code. In industry, many people believe that the migration from traditional software development style based on legacy code maintenance to MDD style is unrealistic. Petre studied how UML (Unified Modeling Language) is used in industry by interviewing with 50 professional software engineers in 50 companies [11]. The results are as follows: *The majority of those interviewed simply do not use UML, and those who do use it tend to do so selectively and often informally.* Fowler points out that people use UML as sketch, blueprint, or programming language [5]. UML as sketch or blueprint is widely accepted in industry. However, UML as programming language is not necessarily adopted. Our approach bridges UML as sketch with UML as blueprint or programming language.

This paper is structured as follows. *Archface*-Centric MDD, a MDD style based on *Archface*, is introduced in Section 2. A method for gradually migrating to *Archface*-Centric MDD is shown in Section 3. Discussion and future work are provided in Section 4.

2. ARCHFACE-CENTRIC MDD

In this section, we introduce *Archface*-Centric MDD [14, 15] in which a design model is treated as a first-class software module termed *design module*.

2.1 Basic Concept

Figure 2 shows the relation among design modules, program modules, and *Archface*, an interface for bridging them. The same *Archface* is modeled by a design model and is implemented by program code. If each type check is correct, a design model is traceable to the code. *Archface* plays a role of design interface for a design model. At the same time, *Archface* plays a role of program interface for code. *Archface* exposes architectural points shared between design and code. These points termed *archpoints* have to be modeled as design points in a UML model and have to be implemented as program points in its code. Class declarations, methods, and events such as *message send* are called design points. An abstraction level—*How much should be a design model more abstract than its code?*—is determined by

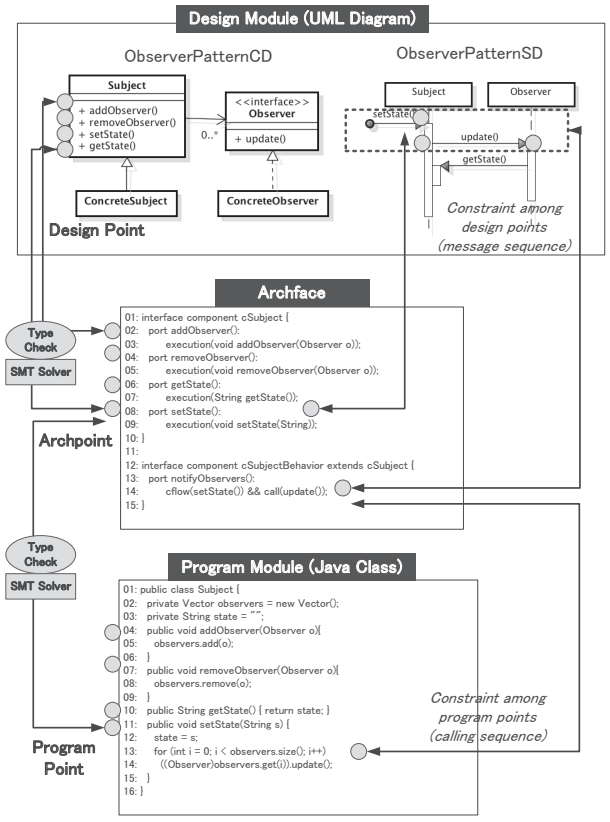

Figure 2: Archface-Centric MDD [15]

selecting archpoints that should be shared between design and code.

Table 1 shows design points, program points, and archpoints. These points can be mapped each other. The idea of archpoints and their selection originates in AOP (Aspect-Oriented Programming) [7]. Archpoints correspond to join points in AspectJ [8]. We focus on archpoints embedded in class and sequence diagrams, because structural and behavioral aspects of software architecture can be basically represented using these diagrams. *Archface* conceptually includes the notion of traditional program interface, because a method definition can be interpreted as a provision of an archpoint selected by an `execution` pointcut in AspectJ.

2.2 Archface, Design and Code

We illustrate design and program modules using the *Observer* pattern, the same example introduced in [15].

2.2.1 Archface

Archface, which supports *component-and-connector* architecture, consists of two kinds of interface, *component* and *connector*. The former exposes archpoints and the latter defines how to coordinate archpoints. Hierarchical definitions are possible, because both interfaces support inheritance. A collaborative architecture can be encapsulated into a group of component and connector interfaces. Pointcut & advice in AspectJ is used as a mechanism for exposing archpoints (pointcut) and coordinating them (advice).

List 1 is a component interface for a subject. *Archface* exposes archpoints from *ports*. Four port declarations (line

Table 1: Design/Program Points and Archpoints

Diagram	Design point (UML2 metamodel)	Program point (Java)	Archpoint (Pointcut)
Class diagram	`Class`	class	a_class (`class`)
(UML)	`Operation`	method	a_method (`method`)
	`Property`	field	a_field (`field`)
Sequence diagram	`Message/MessageEnd` (`sendEvent`)	method call	a_mcall (`call`)*
(UML)	`Message/MessageEnd` (`receiveEvent`)	method exec	a_mexec (`execution`)*
	`Interaction`	(control flow)	(`cflow`)*
Data flow	(`Property` def)	field set	a_def (`set`)*
	(`Property` use)	field get	a_use (`get`)*

*: AspectJ pointcut

02-07) correspond to the traditional interface in which each method declaration can be regarded as exposure of `method execution`. The `notifyObservers` port (line 11-12) exposes an `update call` archpoint that has to be called under the control flow of `setState`. The operator && is used to symbolize *Logical AND*. This archpoint is combined with an `update execution` archpoint specified in a component interface for observers (List 2, line 02).

```
[List 1]
01: interface component cSubject {
02:  port addObserver():
03:       execution(void addObserver(Observer));
04:  port removeObserver():
05:       execution(void removeObserver(Observer));
06:  port getState(): execution(String getState());
07:  port setState(): execution(void setState(String));
08: }
09:
10: interface component cSubjectBehavior extends cSubject {
11:  port notifyObservers():
12:       cflow(setState()) && call(update());
13: }
```

```
[List 2]
01: interface component cObserver {
02:  port update(): execution(void update());
03: }
04:
05: interface component cObserverBehavior extends cObserver {
06:  port updateState():
07:       cflow(update()) && call(String getState());
08: }
```

List 3 is a connector interface specifying the coordination among archpoints exposed from component's ports. The execution of archpoints exposed from component interfaces is coordinated by `connects` (`multiple` indicates the connection is repeatable). In `notifyChange`, an `update call` archpoint in `cSubject` is bound to an `update execution` archpoint in `cObserver`.

```
[List 3]
01: inteface connector cObserverPattern(cSubject, cObserver);
02: inteface connector cObserverPatternBehavior
03:          extends cObserverPattern {
04:  connects multiple notifyChange
05:    (cSubject.notifyObservers, cObserver.update);
06:  connects obtainNewState
07:    (cObserver.updateState, cSubject.getState);
08:  }
09: }
```

2.2.2 Design and Program Modules

Both design and program modules are the same as traditional UML diagrams and code. However, there is a crucial difference. An interface, *Archface*, resides between them and it makes them software modules. `ObserverPatternCD`, a class diagram, and `ObserverPatternSD`, a sequence diagram shown in Figure 2 are design modules faithful to the *Archface* declared in List 1, 2, and 3. A program module is also the same as a traditional module such as a Java class. List 4 and 5 are Java classes implementing the *Archface*.

```
[List 4]
01: public class Subject {
02:  private Vector observers = new Vector();
03:  private String state = "";
04:  public void addObserver(Observer o) {
05:   observers.add(o);
06:  }
07:  public void removeObserver(Observer o) {
08:   observers.remove(o);
09:  }
10:  public String getState() {return state;}
11:  public void setState(String s) {
12:   state = s;
13:   for (int i=0; i<observers.size(); i++)
14:    ((Observer)observers.get(i)).update();
15:  }
16: }
```

```
[List 5]
01: public class Observer {
02:  private subject = new Subject();
03:  private String state = "";
04:  public void update() {
05:   state = subject.getState();
06:   System.out.println("Update received from Subject,
07:       state changed to : " + state);
08:  }
09: }
```

To integrate design and program modules, each design module models its *Archface* and each program module implements the same *Archface*. The conformance to *Archface* can be checked by a type system that takes into account not only program but also design interfaces. The type checking is performed by verifying whether or not a design point (program point) corresponding to an archpoint exists in a design module (program module) while satisfying constraints among design points (program points) (e.g., the order of message sequences specified by `cflow`). Although traditional types are structural—sets of method signatures, *Archface* is based on archpoints including behavior—specified by the order of archpoints because a design model imposes structural or behavioral architectural constraints on a program. The type checking can be implemented using an SMT (Satisfiability Modulo Theories) [3] solver. *Yices* [18] is used as an SMT solver in our compiler implementation, because the order of archpoints can be easily encoded using an array [16].

3. GRADUAL MDD

In this section, we show the notion of gradual MDD using the example in Section 2.

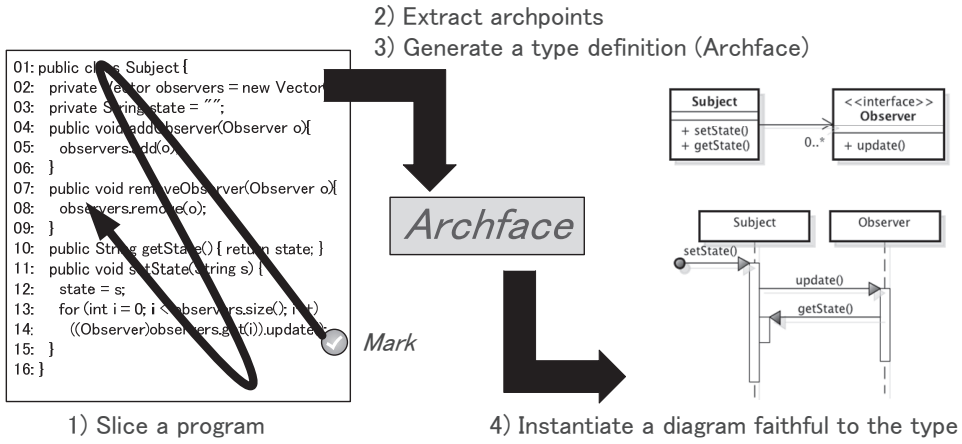

Figure 3: Diagram Generation Process

3.1 Diagram Generation Process

Assume that a developer makes a mark to the `update` call (List 4, line 14) as follows.

```
[From List 4]
14: ((Observer)observers.get(i)).update();  // Mark!
```

The *Archface* definition below is generated from the mark above. The line number corresponds to that of List 1 and 2. We do not have to make marks to all associated program points, because *iArch*, an IDE (Integrated Development Environment) for supporting *Archface*-Centric MDD, provides the archpoints related to the `update` call by slicing the program (List 4, 5). In this case, `a_mcall`/`a_mexec` archpoints of `setState`, `update`, and `getState` are provided as the candidates of the archpoints that should be converted to the corresponding *Archface* definition.

```
// Connector interfaces are omitted due to the space limitation.
[Part of List 1]
01: interface component cSubject {
06:  port getState(): execution(String getState());
07:  port setState(): execution(void setState(String));
11:  port cflow_setState():
12:       cflow(setState()) && call(update());
13: }

[Part of List 2]
01: interface component cObserver {
02:  port update(): execution(void update());
06:  port clow_update():
07:       cflow(update()) && call(String getState());
08: }
```

Figure 3 is a sequential diagram instantiated from this *Archface* definition. *Archface* and a diagram correspond to a type and its instance, respectively. Multiple diagrams can exist corresponding to one type, because a diagram can contain extra design points that are not specified in *Archface*. That is, a diagram can include model elements that are not the target of traceability check. For example, we can describe uncertain design decisions that need not be traceable to the code. The *iArch* IDE generates the most simple diagram whose information is equivalent to the obtained *Archface* definition. This diagram generation can be automated by directly mapping an archpoint in *Archface* to its corresponding design point. The most simple diagram can be a start point of gradual model evolution shown in 3.3, because additional design concerns can be added to this diagram by a developer or newly reversed design concerns can be incrementally reflected to this diagram by the *iArch* IDE.

3.2 Slicing Rules

Table 2 shows the slicing rules for extracting archpoints. Each rule is applied succeedingly without falling in the loop. For example, `Rule2` except `Rule2-2` can be used after `Rule1-2` is applied. However, too many archpoints can be extracted by applying these rules. To deal with this problem, we provide optional rules for eliminating extracted archpoints as follows: OPT1) partial suppress of applying rules in Table 2; OPT2) suppress of succeeding rule application; OPT3) API call/execution elimination; OPT4) constructor elimination; and OPT5) extraction of the longest sequential or data flow. In OPT5, for example, only the message sequence $m_1 \rightarrow m_2 \rightarrow m_3$ is extracted when there are candidates m_1, $m_1 \rightarrow m_2$, and $m_1 \rightarrow m_2 \rightarrow m_3$. By applying these optional rules, we can obtain a simple *Archface* definition that is reasonable for the most developers. In the example shown in 3.1, `Rule4` and succeeding rules are applied to extract a message sequence flow. Java class library calls such as `println` are eliminated by applying optional rule `OPT3`. Moreover, methods such as `addObserver` and `removeObserver` are not contained in Figure 3, because `OPT1` is used—`Rule1-2` is not applied after applying `Rule4-2`.

3.3 Gradual Model Evolution

We can put MDD into practice using a generated design model. We can modify the model, reflect it to the *Archface*, and re-generate the code while preserving the other part of the existing code. The *iArch* compiler embeds the code snippets corresponding to the updated *Archface* into the existing code. If a developer adds a new message sequence in a design model, the *iArch* compiler generates a control flow of the associated method calls/executions and just appends them to the existing code. Our approach does not need annotations for this kind of round-trip-engineering. *Archface* plays an important role in preserving the traceability without breaking an abstraction level. On the other hand, if a developer considers that unmarked methods such as `addObserver` and

Table 2: Slicing Rules for Extracting Archpoints

No.	Original Archpoint		Inferred Archpoints (Candidates)
1.	a_class	1-1)	corresponding `a_class`
		1-2)	`a_method` contained in `a_class`
		1-3)	`a_field` contained in `a_class`
2.	a_method	2-1)	corresponding `a_method`
		2-2)	`a_class` containing `a_method`
		2-3)	`a_field` accessed from `a_method`
		2-4)	`a_mcall` calling `a_method`
		2-5)	`a_mexec` executing `a_method`
3.	a_field	3-1)	corresponding `a_field`
		3-2)	`a_class` containing `a_field`
		3-3)	`a_method` accessing `a_field`
		3-4)	`a_def` defining `a_field`
		3-5)	`a_use` using `a_field`
4.	a_mcall	4-1)	corresponding `a_mcall`
		4-2)	`a_class` containing `a_mcall`
		4-3)	`a_method` containing `a_mcall`
		4-4)	`a_mexec` called by `a_mcall`
5.	a_mexec	5-1)	corresponding `a_mexec`
		5-2)	`a_class` containing `a_mexec`
		5-3)	`a_method` containing `a_mexec`
		5-4)	`a_mcall` executed by `a_mexec`
6.	a_def	6-1)	corresponding `a_def`
		6-2)	`a_class` containing `a_def`
		6-3)	`a_method` containing `a_def`
		6-4)	`a_field` defined by `a_def`
7.	a_use	7-1)	corresponding `a_use`
		7-2)	`a_class` containing `a_use`
		7-3)	`a_method` containing `a_use`
		7-4)	`a_field` used by `a_use`

`removeObserver` should be reflected to a design model, he or she only has to make the marks to these method definitions (List 4, line 04, 07). A new design model generated from additional marks is merged with the existing model. The design model is gradually improved as the knowledge of the existing code increases.

4. DISCUSSION AND FUTURE WORK

As one of the important research directions in the field of software design and architecture, Taylor et al. pointed out the need for adequate support for fluidly moving between design and coding tasks [13]. Gradual MDD is a solution to this issue. Our approach guarantees the traceability between design and code by introducing the notion of *Archface*.

There are several studies on design traceability. Aldrich et al. proposed *ArchJava* [2], an extension of Java. ArchJava unifies architecture and implementation, ensuring that the implementation conforms to architectural constraints. *Umple* [17] supports the notion of model-oriented programming that adds modeling features derived from UML to object-oriented languages such as Java. Using *ArchJava* or *Umple*, we can merge modeling with programming. Cassou et al. explored the design space between abstract and concrete component interaction specifications [4]. Zheng and Taylor proposed 1.x-way architecture-implementation mapping [19] for deep separation of generated and non-generated code. On the other hand, in our approach, a type-based module integration mechanism plays a key role in bridging design and code while preserving an abstraction level. JaMoPP, a set of plug-ins for parsing Java code into models based on EMF (Eclipse Modeling Framework), bridges the gap between modelling and programming [6]. MoDisco is a framework to develop model-driven tools supporting software modernization [10]. Both JaMoPP and MoDisco provide reverse engineering facilities. Our approach focuses on reverse engineering taking abstraction into account.

We are developing the *iArch* IDE consisting of 1) model & program editor, 2) *Archface* generator, 3) abstraction-aware compiler, and 4) abstraction metrics calculation. Program slicing for extracting archpoints is under construction. This IDE implemented as an Eclipse plug-in using EMF supports class and sequential diagrams whose metamodels are basically the same as UML2 ecore metamodels. The syntax of *Archface* is based on AspectJ pointcuts and is slightly complex. To relax this problem, the *iArch* IDE introduces syntactical sugar consisting of a Java-like interface (structural part) and an LTS (Labelled Transition Systems) notation (behavioral part). We plan to implement *iArch* on a tablet PC to realize a sketch-like user interface.

We take a hint from *gradual typing* [12] in which a type may be determined either at compile-time (static typing) or at run-time (dynamic typing). *Archface* is a type between design and code. Forward and reverse engineering in MDD can be considered *model-time typing* and *review-time typing*, respectively. A reasonable *Archface* can be obtained by gradually performing both kinds of typing.

The empirical evaluation using large scale code-bases is a crucial issue. We plan to apply our idea to generate an abstract design model from open source software repositories.

5. ACKNOWLEDGMENTS

This research is being conducted as a part of the Grant-in-aid for Scientific Research (A) 26240007 and Challenging Exploratory Research 25540025 by the Ministry of Education, Culture, Sports, Science and Technology, Japan.

6. REFERENCES

[1] Ai, D., Ubayashi, N., Li, P., Hosoai, S., Kamei, Y.: iArch: An IDE for Supporting Abstraction-aware Design Traceability, 2nd International Conference on Model-Driven Engineering and Software Development (MODELSWARD 2014), pp.442-447, 2014.

[2] Aldrich, J., Chambers, C., and Notkin, D.: ArchJava: Connecting Software Architecture to Implementation, 24th International Conference on Software Engineering (ICSE 2002), pp.187-197, 2002.

[3] Biere, A., Heule, M., Maaren, H. V., and Toby Walsh, T.: *Handbook of Satisfiability*, Ios Pr Inc, 2009.

[4] Cassou, D., Balland, E., Consel, C., and Lawall, J.: Leveraging Software Architectures to Guide and Verify the Development of Sense/Compute/Control Applications, 33rd International Conference on Software Engineering (ICSE 2011), pp.431-440, 2011.

[5] Fowler, M.: *UML Distilled*, Addison-Wesley, 2003.

[6] JaMoPP, http://www.jamopp.org/.

[7] Kiczales, G., Lamping, J., Mendhekar A., Maeda, C., Lopes, C., Loingtier, J. and Irwin, J.: Aspect-Oriented Programming, In *Proceeding of European Conference on Object-Oriented Programming (ECOOP'97)*, pp.220-242, 1997.

[8] Kiczales, G., Hilsdale, E., Hugunin, J., et al.: An Overview of AspectJ, In *Proceedings of European Conference on Object-Oriented Programming (ECOOP 2001)*, pp.327-353, 2001.

[9] Kramer, J.: Is Abstraction the Key to Computing? Communications of the ACM, Vol. 50 Issue 4, pp.36-42, 2007.

[10] MoDisco, http://www.eclipse.org/MoDisco/.

[11] Petre, M.: UML in Practice, 35th International Conference on Software Engineering (ICSE 2013), pp.722-731, 2013.

[12] Siek, J. G. and Taha, W.: Gradual Typing for Object, 21st European Conference on Object-Oriented Programming (ECOOP 2007), pp.2-27, 2007.

[13] Taylor, R. N. and Hoek, A.: Software Design and Architecture –The once and future focus of software engineering, 2007 Future of Software Engineering (FOSE 2007), pp.226-243, 2007.

[14] Ubayashi, N., Nomura, J., and Tamai, T.: Archface: A Contract Place Where Architectural Design and Code Meet Together, 32nd International Conference on Software Engineering (ICSE 2010), pp.75-84, 2010.

[15] Ubayashi, N. and Kamei, Y.: Design Module: A Modularity Vision Beyond Code, 5th International Workshop on Modelling in Software Engineering (MiSE 2013, Workshop at ICSE 2013), pp.44-50, 2013.

[16] Ubayashi, N., Ai, D., Li, P., Li, Y., Hosoai, S., and Kamei, Y.: Abstraction-aware Verifying Compiler for Yet Another MDD, In *Proceedings of the 29th International Conference on Automated Software Engineering (ASE 2014)*, New Ideas Paper, to appear, 2014.

[17] Umple: http://cruise.eecs.uottawa.ca/umple/.

[18] Yices: http://yices.csl.sri.com/

[19] Zheng, Y. and Taylor, R. N.: Enhancing Architecture-Implementation Conformance with Change Management and Support for Behavioral Mapping, 34th International Conference on Software Engineering (ICSE 2012), pp.628-638, 2012.