Architectural Point Mapping for Design Traceability

Naoyasu Ubayashi
Kyushu University
Fukuoka
Japan
ubayashi@acm.org

Yasutaka Kamei
Kyushu University
Fukuoka
Japan
kamei@ait.kyushu-u.ac.jp

Abstract

AOP can be applied to not only modularization of crosscutting concerns but also other kinds of software development processes. As one of the applications, this paper proposes a design traceability mechanism originating in join points and pointcuts. It is not easy to design software architecture reflecting the intention of developers and implement the result of design as a program while preserving the architectural correctness. To deal with this problem, we propose two novel ideas: *Archpoint (Architectural point)* and *Archmapping (Archpoint Mapping)*. Archpoints are points for representing the essence of architectural design in terms of behavioral and structural aspects. By defining a set of archpoints, we can describe the inter-component structure and the message interaction among components. Archmapping is a mechanism for checking the bidirectional traceability between design and code. The traceability can be verified by checking whether archpoints in design are consistently mapped to program points in code. For this checking, we use an SMT (Satisfiability Modulo Theories) solver, a tool for deciding the satisfiability of logical formulas. The idea of archpoints, program points, and their selection originates in AOP.

Categories and Subject Descriptors: D.2.11 Software Engineering: Software Architectures—Languages

General Terms: Design, Verification

Keywords: Design traceability, SMT solver

1. Introduction

AOP can be applied to not only modularization of crosscutting concerns but also other kinds of software development processes. As one of the applications, this paper proposes a design traceability mechanism in which the essential idea originates in join points and pointcuts.

Although architectural design plays an important role in software development, it is not easy to design architecture reflecting the intention of developers and implement the result of design as a program while preserving architectural correctness and adequate abstraction level, because there is a gap between design and code.

To deal with this problem, we propose two novel ideas: *Archpoint (Architectural point)* and *Archmapping (Archpoint Mapping)*. The purpose of these ideas is to describe software design based on the *component-and-connector* architecture [2] and verify the traceability between design and its implementation. Archpoints are points for representing the essence of architectural design in terms of behavioral and structural aspects. By defining a set of archpoints, we can describe the inter-component structure and the message interaction among components. Archmapping is a mechanism for checking the design traceability. An archpoint such as *message send* in design is mapped to a program point such as *method call* in code. All program points are not associated to archpoints because architectural design should be abstract and the detailed considerations about implementation should not be included in the design. Archpoints can be considered as selected program points that should be shared between design and code. The idea of archpoints, program points, and their selection originates in AOP notion such as join points and pointcuts. The traceability can be verified by checking whether archpoints are consistently mapped to program points. This mapping is bidirectional. For this checking, we use an SMT (Satisfiability Modulo Theories) solver [4], a tool for deciding the satisfiability of logical formulas. SMT generalizes SAT (Satisfiability) [4] by adding equality reasoning, arithmetic, and other first-order theories. The properties of archpoints and program points are encoded to logical formulas and checked by an SMT solver.

The remainder of this paper is structured as follows. In Section 2, we point out the problems concerning design traceability. In Section 3, the notion of *Archpoint* and *Archmapping* is introduced. In Section 4, SMT-based veri-

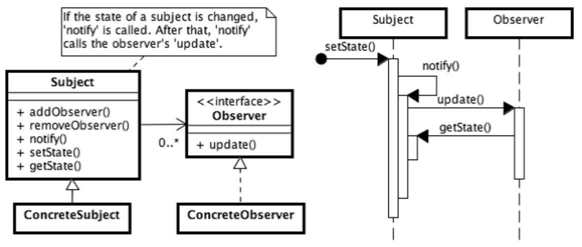

Figure 1. Observer pattern described in UML

fication is illustrated. Related work and concluding remarks are provided in Section 5.

2. Motivation

In this section, we point out what kinds of problems occur between design and code by using an example.

2.1 Design traceability

The *Observer* pattern, one of the GoF design patterns, is convenient for discussing the problems between design and code, because the pattern not only has architectural characteristics such as collaboration but also is relatively close to implementation. The *Observer* pattern consists of a `Subject` and an `Observer`. When the state of a subject is changed, the subject notifies all observers of this new state.

Figure 1 illustrates the *Observer* pattern described in UML (Unified Modeling Language). Design models can be represented by using class diagrams, interaction diagrams, and so on. The note in Figure 1 shows that `notify` should be called under the control flow of `setState`.

Although UML is easy to read and understand, it is not easy to write a program consistent with the design intent because it tends to be informally described.

List 1 is a program written by a novice. The class structure conforms to the design model and this program behaves correctly. However, List 1 does not conform to the note in Figure 1 because `notify` is not called.

```
[List 1]
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 void notify() {
11:     for (int i = 0; i < observers.size(); i++)
12:       ((Observer)observers.get(i)).update();
13:   }
14:   public String getState() { return state; }
15:   public void setState() {
16:     state = s;
17:     for (int i = 0; i < observers.size(); i++) // code clone
18:       ((Observer)observers.get(i)).update();
19:   }
20: }
21:
22: public class Observer {
23:    private subject = new Subject();
24:    private String state = "";
25:    public void update() {
26:      state = subject.getState();
27:      System.out.println("Update received from Subject,
28:        state changed to : " + state);
29:    }
30: }
```

In List 1, there is a code clone (line 11 - 12, line 17 - 18). A code clone tends to occur while debugging. It is not easy for most programmers to be aware that they violate the intent of architectural design because their programs successfully execute even if there is a code clone. List 2 is an implementation conforming to the design.

```
[List 2]
public void setState() {
  state = s;
  notify();
}
```

There is another problem in List 1. Although List 1 includes libraries such as `Vector`, `add`, `remove`, and `println`, they do not appear in the design model. Should we reflect these elements in the model ? Our answer is NO because software architecture should be abstract and include only the essence of design intent.

Next, assume that a developer changes the old code to a new version in which `notify` is not called directly but a method is called from `setState` and the method calls `notify`. In this case, the design model has only constraints such that `notify` is called under the control flow of `setState`. Thus, the design model should not be changed even if the code is modified. A design model can be related to multiple code implementations.

2.2 Current MDD technology

Although many MDD (Model-Driven Development) tools can generate code from UML diagrams, most of them generate only skeleton code. A developer then must add code to the auto-generated code and in doing so might make a mistake. Interaction diagrams are useful for describing behavioral design intents. For example, an interaction diagram can represent such a design intent that `notify` should be called under the control flow of `setState`. However, it is not necessarily easy to generate code from interaction diagrams because one interaction can be mapped to a variety of code. Moreover, it is so difficult to maintain the traceability between interaction diagrams and code with preserving adequate abstraction level. Although it is relatively easy to generate code from state machine diagrams, the maintenance of traceability is not easy due to the same reason.

The developer might create a detailed model using action semantics to generate full code from UML diagrams. However, the contents of the diagrams are semantically equivalent to the code. This violates a principle of abstractions required in architectural design as mentioned above. A design model should be at an adequately abstract level, not at the same level as code.

2.3 Problems to be tackled

Problems between design and code can be summarized as follows: 1) It is not easy to reflect the design decisions at the code level; and 2) It is not easy to synchronize design and code with preserving adequate abstraction level. These

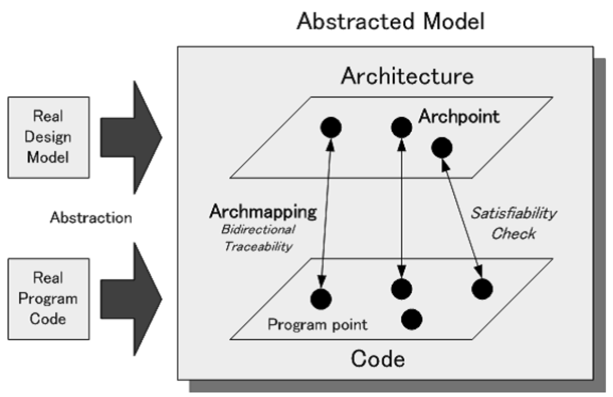

Figure 2. Archpoint and Archmapping

Category	Archpoint	Program point (Java)
Class diagram	class	class definition
(UML)	method	method definition
	field	variable definition
Interaction diagram	message send	method call
(UML)	message receive	method execution
Data flow	def	field set
	use	field get

Table 1. Archpoint and program points (a part)

problems indicate that a mechanism for checking the design traceability is needed.

3. Archpoint and Archmapping

To deal with the problems in Section 2, two novel ideas *Archpoint* and *Archmapping* are provided.

3.1 Basic concept

Figure 2 illustrates the concept of *Archpoint* and *Archmapping*. Archpoints are points for describing the essence of architectural design at the adequate abstraction level. The notion of archpoints is similar to that of join points (or program points) in AOP.

Table 1 shows major archpoints. In general, software architecture is represented by structural and behavioral aspects. The former can be modeled by class diagrams, and the latter can be represented by interaction diagrams. Here, for simplicity, some features including object instantiation and inheritance are omitted in Table 1.

As mentioned before, archpoints can be considered as selected program points that should be shared between design and code. So, an archpoint can be mapped to a program point. Archmapping is a mechanism for this purpose. Table 1 shows a mapping in case of Java.

As illustrated in Figure 2, in our approach, an abstract model of architectural design is represented by archpoints and constraints among them. In the same way, an abstract model of a program is also represented by program points and constraints among them. The synchronization (or traceability) between design and code is maintained by bidirectionally mapping archpoints and corresponding program points. We can preserve adequate abstraction level by ignoring other program points that are not associated to archpoints as shown in Figure 3. The execution order of archpoints representing the *Observer* pattern is preserved even if program points, which do not change the sequence of archpoints, are added to the code. In other words, the traceability between design and code can be maintained if there is a bisimulational relation between them in terms of archpoints and added program points do not violate this bisimulation. The constraints such as execution order of archpoints are encoded to logical formulas. The traceability can be verified by checking the satisfiability of the logical formulas as mentioned below.

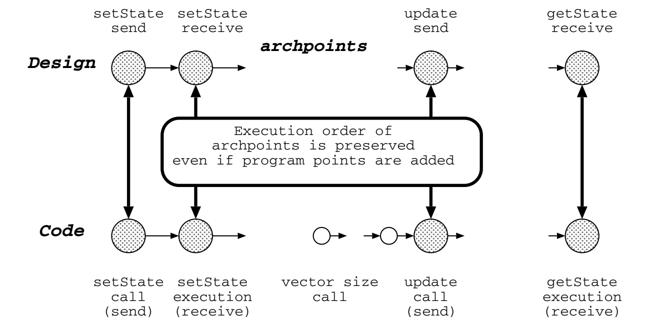
setState send
setState receive
archpoints
update send
getState receive
Design
Execution order of archpoints is preserved even if program points are added
Code
setState call (send)
setState execution (receive)
vector size call
update call (send)
getState execution (receive)

Figure 3. Bidirectional traceability

3.2 Design description

Architecture is define as a set of archpoints $A = A_1, ..., A_n$ and a set of constraints among them. Design is regarded correct if the logical formula below is satisfied. $Archcond_i$ is a logical expression for specifying a property that should be satisfied at a set of related archpoints.

$$ARCHITECTURE = archcondA_1 \wedge ... \wedge archcondA_m \quad (1)$$

In case of the *Observer* pattern, a part of architecture (notification sequence) can be described below. `Message_sequence` is a predicate that is satisfied when the order of archpoint occurrence is correct. `Message_iteration` is a predicate showing iteration. By defining a set of predicates such as *inheritance_relation*, *control_flow*, and *data_flow*, we can describe a variety of architectural properties.

```
Observer_Pattern :=
  message_sequence(                          ; [predicate]
    cSubject_setState_message_send,          ;  archpoint
    cSubject_setState_message_receive,       ;  archpoint
    cSubject_notify_message_send,            ;  archpoint
    cSubject_notify_message_receive,         ;  archpoint
    massage_iteration(                       ; [predicate]
      cObserver_update_message_send,         ;  archpoint
      cObserver_update_message_receive,      ;  archpoint
      cSubject_getState_message_send,        ;  archpoint
      cSubject_getState_message_receive))    ;  archpoint
```

3.3 Program description

A program can be abstracted as a set of program points $P = P_1, ..., P_{n'}$ and a set of constraints among them. An implementation is consistent if the logical formula below is satisfied. $Progcond_i$ is a logical expression for specifying a property that should be satisfied in a set of program points.

$$PROGRAM = progcondP_1 \wedge ... \wedge progcondP_{m'} \quad (2)$$

In case of List 1, the behavioral aspect can be described below. `Calling_sequence` is a predicate specifying the calling sequence of methods. `Calling_iteration` is a predicate showing iteration.

```
Program_List1 :=
  calling_sequence(                            ; [predicate]
    cSubject_setState_call,                    ;  program point
    cSubject_setState_execution,               ;  program point
    calling_iteration(                         ; [predicate]
      Vector_size_call,                        ;  program point
      Vector_size_execution,                   ;  program point
      Vector_get_call,                         ;  program point
      Vector_get_execution,                    ;  program point
      cObserver_update_call,                   ;  program point
      cObserver_update_execution,              ;  program point
      cSubject_getState_call,                  ;  program point
      cSubject_getState_execution,             ;  program point
      System_out_println_call,                 ;  program point
      System_out_println_execution))           ;  program point
```

3.4 Archmapping for traceability

A refinement mapping from an architectural design to the code can be defined as a mapping function `refine`. In case of the *Observer* pattern, a part of refinement mapping can be defined below. The predicates `message_sequence` and `message_iteration` should be also mapped to `calling_sequence` and `calling_iteration`, respectively.

```
refine( cSubject_setState_message_send ) =
  cSubject_setState_call
refine( cSubject_setState_message_receive ) =
  cSubject_setState_execution
```

The refinement is correct if the following is satisfied.

$$refine(ARCHITECTURE) \wedge PROGRAM \quad (3)$$

In case of the *Observer* pattern, the logical formula $refine(Observer\_Pattern) \wedge Program\_List1$ can be described as follow. In this case, the formula is not satisfied because the first `calling_sequence` is false (`notify` is not called and executed). That is, List 1 does not conform to the architectural design (*Observer* pattern).

```
calling_sequence(   ; not satisfied (mapped from Observer_Pattern)
  cSubject_setState_call,
  cSubject_setState_execution,
  cSubject_notify_call,
  cSubject_notify_execution,
  calling_iteration(
    cObserver_update_call,
    cObserver_update_execution,
    cSubject_getState_call,
    cSubject_getState_execution))
  ∧
calling_sequence(  ; satisfied (List 1)
  cSubject_setState_call,
  cSubject_setState_execution,
  calling_iteration(
    Vector_size_call,
    Vector_size_execution,
    Vector_get_call,
    Vector_get_execution,
    cObserver_update_call,
    cObserver_update_execution,
    cSubject_getState_call,
    cSubject_getState_execution,
    System_out_println_call,
    System_out_println_execution))
```

Table 2. Yices input language (a part)

Language construct	Syntax
Type definition	(define-type [name] [type])
Constant definition	(define [name]::[type] [expr])
Basic types	real, int, nat, bool
Subtype	(subtype ([id]::[type]) [expr])
Sub range	(subrange expr$_l$ expr$_u$)
Boolean operators	(and [expr$_1$] ... [expr$_n$])
	(or [expr$_1$] ... [expr$_n$])
	(not [expr])
Equality	(= [expr$_1$] [expr$_2$])
Disequality	(/= [expr$_1$] [expr$_2$])
Arithmetic	$<, <=, >, >=, +, -, *, /$, div, mod
Assert	(assert [expr])
Check	(check)

On the other hand, List 2 conforms to the design because the first `calling_sequence` is satisfied. `Calling_sequence` is true if the program points specified in the arguments are in order. In architectural design, we do not have to consider the existence of `System_out_println_call` and `System_out_println_execution` because architecture should be abstract. So, architecture does not have to be modified even if `println` is removed from List 2 because the first `calling_sequence` remains true. As mentioned here, the bidirectional traceability between design and code can be maintained with preserving the adequate abstraction level.

4. SMT-based traceability check

We are developing an SMT-based support tool that automates the traceability check. We use *Yices* [9] as an SMT solver. We plan to develop a tool consisting of three features: automatic archpoints extraction from UML design models, automatic program points (shadows) extraction from Java programs, and encoding to the *Yices* input language.

In this section, the overview of our approach is illustrated in terms of *Yices* encoding.

4.1 Yices

Yices provides an input language whose syntax is similar to Scheme as shown in Table 2. *Yices* decides the satisfiability of formulas containing uninterpreted function symbols with equality, linear real and integer arithmetic, scalar types, recursive datatypes, tuples, records, extensional arrays, fixed-size bit-vectors, quantifiers, and lambda expressions. *Yices* is effective for traceability check mentioned in Section 3 because these expressive logical formulas can be used.

4.2 Yices encoding

The formula $refine(Observer\_Pattern) \wedge Program\_List1$ can be encoded to List 3.

```
[List 3]
01: (define-type_count (subrange 0 11)) ; 0<= count <= 11
02: (define i0::count)
03:   ...
04: (define i7::count)
05:
06: (assert (and                        ; assertion
07: ;; refine(Observer_Pattern)
08:   (< i0 i1) (< i1 i2) (< i2 i3) (< i3 i4)
09:    (< i4 i5) (< i5 i6) (< i6 i7)
10:    (= (list1 i0) cSubject_setState_call)
11:    (= (list1 i1) cSubject_setState_execution)
12:    (= (list1 i2) cSubject_notify_call)
13:    (= (list1 i3) cSubject_notify_execution)
14:    (= (list1 i4) cObserver_update_call)
15:    (= (list1 i5) cObserver_update_execution)
16:    (= (list1 i6) cSubject_getState_call)
17:    (= (list1 i7) cSubject_getState_execution)
18: ;; Program_List1
19:    (= (list1  0) cSubject_setState_call)
20:    (= (list1  1) cSubject_setState_execution)
21:         :
22:    (= (list1 11) System_out_println_execution)))
23:
14: (check)                               ; check the assertion
```

The symbol `list1`, whose definition is omitted due to the space limitation, is an array including all program points in List 1. The occurrence order of refine(archpoint) specified in `calling_sequence` and `calling_iteration` is encoded in line 08 - 17. The predicate `calling_iteration` can be encoded to *Yices* by expanding the iteration limited times (one time in List 3). In this case, only the bounded checking is available. As shown in List 3, predicates representing architectural constraints can be compiled into the *Yices* input language.

4.3 Traceability check

The assertion in List 3 is not satisfied because line 12 - 13 is not satisfied. As demonstrated here, we can automatically check the design traceability by using *Yices*.

Our approach can be used as a bounded model checker for verifying temporal behavior of architectural design.

For example, a temporal specification

```
cSubject_setState_message_receive
        -> <>cObserver_getState_message_send
```

can be checked. <> (in the future) is an operator of LTL (Linear Temporal Logic). The meaning of the formula is as follow: `getState` message will be sent from an observer in the future if `setState` message is received in a subject. This LTL formula can be encoded to List 4. The symbol `alist` is an array including all archpoints in Figure 1.

```
[List 4]
(assert (and
  (< i j)
  (= (alist i) cSubject_setState_message_receive)
  (= (alist j) cObserver_getState_message_send)))
```

This assertion is satisfied. *Observer* pattern in Figure 1 is designed correctly in terms of the above specification. In the above case, we can regard that this temporal specification is correctly implemented as program code (List 2), because both of architectural design and its refinement are correct.

5. Discussion and Future work

There are several attempts to unify architecture, code, and components. ArchJava [1] ensures that an implementation conforms to architectural constraints. *Archface* [8] separates architecture definitions from an actual implementation by introducing a new interface mechanism. Mezini et al. proposed adaptive plug and play components called aspectual components [6]. Eichberg et al. proposed declarative queries for grouping source elements across programming language module boundaries, because dependencies between program elements need to be modeled from different perspectives reflecting architectural, design, and implementation level decisions [5]. Bagheri et al. showed a way for automated formal derivation of style-specific architectures [3]. An application model should be mapped to one or more architectures.

Archface plays a role as an ADL (Architecture Description Language) at the design phase and as a programming interface at the implementation phase. The result of the design modeling is stored in the form of *Archface* (ADL). After that, a program preserving the architectural intention is developed by implementing the *Archface* (programming interface). One may think that it is not easy to support *Archmapping* because it is difficult to automatically extract program points and associate them with archpoints. Adopting *Archface*, this task becomes easy because program points shared between design and code are explicitly declared using AspectJ-like pointcuts such as `call` and `cflow`.

In this paper, we considered only *one-to-one mapping* between archpoints in design and program points in code. We plan to support *one-to-multiple mapping* to model not only code generation (forward traceability) but also design recovery (backward traceability) more practically. We think that we have to introduce the notion of a virtual program point consisting of real program points in order to convert *one-to-multiple mapping* to *one-to-one mapping*.

Acknowledgement

This research is being conducted as a part of the Grant-in-aid for Scientific Research (B), 23300010 by the Ministry of Education, Culture, Sports, Science and Technology, Japan.

References

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

[2] Allen, R. and Garlan, D.: Formalizing Architectural Connection, In *Proceedings of the 16th International Conference on Software Engineering (ICSE'94)*, pp.71-80, 1994.

[3] Bagheri, H., Song, Y., and Sullivan, K. J. : Architectural Style as an Independent Variable, In *Proceedings of the 25th IEEE/ACM International Conference on Automated Software Engineering (ASE 2010)*, pp.159-162, 2010.

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

[5] Eichberg, M., Kloppenburg, S., Klose, K., and Mezini, M.: Defining and Continuous Checking of Structural Program Dependencies, In *Proceedings of the 30th ACM/IEEE International Conference on Software Engineering (ICSE 2008)*, pp.391-400, 2008.

[6] Mezini, M. and Lieberherr, K.: Adaptive Plug-and-play Components for Evolutionary Software Development, In *Proceedings of the 13th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications (OOPSLA '98)*, pp.97-116, 1998.

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

[8] Ubayashi, N., Nomura, J., and Tamai, T.: Archface: A Contract Place Where Architectural Design and Code Meet Together, In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering (ICSE 2010)*, pp.75-84, 2010.

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