Korea-Japan Joint Workshop on ICT (Pohang, Korea, 20-22 September 2012), Paper #14

Archpoint and Archmapping —Bidirectional Traceability between Design and Code—

Naoyasu Ubayashi[1,a)] Yasutaka Kamei[1,b)]

Abstract: Architecture plays an important role in software development. Although well-designed architecture leads to high-quality systems, 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 properties of archpoints and program points are encoded to logical formulas and checked by an SMT solver.

Keywords: Architecture, bidirectional traceability, architectural point, SMT solver

1. Introduction

Architectural design plays an important role in software development because system characteristics such as robustness and maintainability depend on the architecture. Well-designed architecture leads to high-quality systems.

However, 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, because there is a gap between design and implementation. 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 [4].

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 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 [3], a tool for deciding the satisfiability of logical formulas. SMT generalizes SAT (Satisfiability) [3] 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 verification is illustrated. In Section 5, research challenges are discussed. Concluding remarks are provided in Section 6.

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, sequential diagrams, and so on. The note

[1] Kyushu University, 744 Motooka, Nishi-ku, Fukuoka 819–0395, Japan
[a)] ubayashi@acm.org
[b)] kamei@ait.kyushu-u.ac.jp

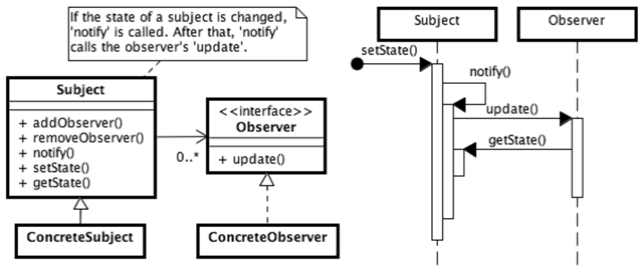

Fig. 1 Observer pattern described in UML

in the class diagram and the sequential diagram show 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.

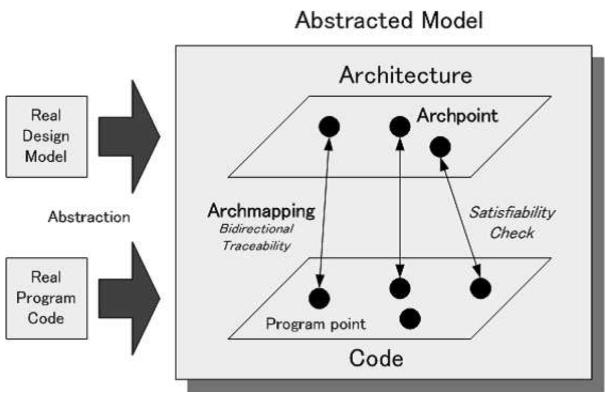

Fig. 2 Archpoint and Archmapping

Category	Archpoint	Program point (Java)
Class diagram	class	class definition
(UML)	method	method definition
	field	variable definition
Sequential 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)

2.2 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 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.

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 sequential 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. The constraints are encoded to logical formulas. The traceability can be verified by checking the satisfiability of the logical formulas as mentioned below.

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 predicates such as *inheritance_relation* and *control_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. `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))
```

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. 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* [6] 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 and Lisp. *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. 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.

```
[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 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*.

In this paper, we discussed the traceability from the viewpoint of a component interaction represented by a message sequence. There are other architectural aspects such as class structure and data flow. These aspects can be also encoded in *Yices* and can be verified by its solver. These encoding rules can be categorized into several patterns corresponding to architectural description types.

5. Research challenges

In this section, we show research challenges towards verifiable design traceability.

Challenge 1: Verifiable architectural interface

There are several attempts to unify architecture and code. For example, ArchJava [1] ensures that the implementation conforms to architectural constraints. *Archface* [5] enhances this approach and separates architecture definitions from actual implementation by introducing a new interface mechanism. *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 architectural 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). *Archface* can be considered a kind of contract between design and implementation.

Although *Archface* is one of the promising approaches that bridge design and code, the traditional type systems are insufficient for inconsistency check because architectural interfaces contain rich information that may not be able to syntactically checked. Most type systems only check the statical aspect of programming languages. It is an important research challenge to provide verifiable architectural interface mechanisms. We think that *Archface* can be translated into a set of archpoints and constraints among them.

Someone 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 in *Archface*. Using this declaration information, program points associated to archpoints can be easily extracted by code analysis.

Challenge 2: Common traceability framework

There are a variety of checking such as *"traceability between design and code"* and *"traceability between design and execution"*. Some kind of common framework is needed to check properties including structure, behavior, implementation, and testing. In traditional approaches, these properties are checked by using independent tools. For example, the behavioral aspect is checked by model checkers and the traceability between design and code is checked by testing. These checking activities are separated and their theoretical bases are different each other. An SMT-based approach has a possibility of integrating these checking activities. The behavioral aspect of architecture design can be checked by a bounded model checker based on SMT. The traceability between design and code can be checked by the method proposed in this paper. The traceability between design and execution can be checked by logging the program execution and verifying the traced data using an SMT solver.

6. Conclusion

This paper proposed two new ideas: *Archpoint* and *Archmapping*. Our approach is the fruitful integration of a design abstraction mechanism based on archpoints, bidirectional mapping between archpoints and program points, and SMT-based verification. As mentioned in Section 5, our approach can be extended to a variety of research fields because SMT is simple and strong.

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] Biere, A., Heule, M., Maaren, H. V., and Toby Walsh, T.: *Handbook of Satisfiability*, Ios Pr Inc, 2009.
pp.81 - 94, 2006.

[4] 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.

[5] 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.

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