Archface: A Contract Place Where Architectural Design and Code Meet Together

Naoyasu Ubayashi Jun Nomura
Kyushu Institute of Technology
Fukuoka, Japan
{ubayashi,nomura}@minnie.ai.kyutech.ac.jp

Tetsuo Tamai
University of Tokyo
Tokyo, Japan
tamai@acm.org

ABSTRACT

This paper proposes *Archface*, an interface mechanism for bridging the gap between architectural design and its implementation. *Archface*, which encapsulates the essence of architectural design, is not only an architecture description language (ADL) but also a programming-level interface. *Archface* is based on the *component-and-connector* architecture, one of the most popular architectural styles. *Archface* is effective for software evolution because the traceability between design and its implementation can be realized by enforcing architectural constraints on the program implementation. This traceability is bidirectional. *Archface* provides a place where design and code meet together. In *Archface*, a component exposes program points such as method call/execution and a connector defines how to coordinate exposed program points. This mechanism is based on aspect orientation. A collaborative architecture consisting of components can be encapsulated into a group of interfaces and separated from implementation because dynamic program points representing control flow can be specified in the interfaces. We can characterize the notion of *Archface* with a simple word *"predicate coordination"* in which program points are exposed by a predicate (pointcut) and coordinated each other by a trait-based connector (advice). *Archface* facilitates not only bidirectional traceability but also architectural reuse, composition, and verification.

Categories and Subject Descriptors: D.2.11 Software Architectures: Language

General Terms: Design

Keywords: Architecture, ADL, Interface, Co-evolution, Bidirectional Traceability, Predicate Coordination

1. INTRODUCTION

Architectural design plays an important role in the software development because system characteristics such as robustness, reliability, and maintainability depend on software architecture. Bass, L., et al. defined software architecture [4]: *The software architecture of a program or computing system is the structure or structures of the system, which comprise of software elements, the externally visible properties of those elements, and relationships among them.* Well-designed architecture leads to high quality systems. A variety of architecture description languages (ADL) have been proposed in order to support architectural design. An ADL enables us to define software architecture rigorously and verify their consistency and correctness.

However, it is not easy to design software architecture reflecting the intention of developers and implement the result of design modeling as a program preserving the architectural correctness because there is the gap between design and implementation. Taylor, R. N., et al. provided important research directions in the field of software design and architecture [26]. As one of the issues to be tackled, they pointed out the adequate support for moving from architecture to implementation and fluidly moving between design and coding tasks. Most of the current model-driven development (MDD) tools can generate only code skeletons from design models. To generate full code, we have to adopt domain-specific approaches in which a sufficient set of libraries are prepared for code generation or we have to create such models that contain detailed implementation-level behavioral specifications. Although the former is effective, it is not necessarily easy to construct such environments. The latter is not favorable because design models should be abstract and contain only the essential aspects of architectural design decisions. The detailed consideration about implementation should not be included in the design models. In the case of the skeleton code, a programmer has to write the rest of the code and might make a mistake that violates the architectural correctness. However, the defects embedded in the code cannot be automatically detected because there is no language-level traceability between architectural design and its implementation. This traceability should be bidirectional. A change in a design model should be transferred to its implementation. Moreover, a change in the code should be also reflected on the corresponding design model. Unfortunately, current MDD tools are insufficient to realize this kind of bidirectional traceability.

To deal with this problem, we propose *Archface*, a new interface mechanism that takes into account the importance of architecture and integrates an architectural design model with its implementation. There are a variety of architectural styles and patterns according to the target applications, scope, and purposes. *Archface* is based on the *component-and-connector* architecture [3], one of the most popular architectural styles. For example, most of the POSA (Pattern-Oriented Software Architecture) [5] patterns, a famous architectural catalogue, can be represented by the *component-and-connector* architecture. In the POSA patterns, collaboration among components is important because it determines the system behavior.

Archface plays a role as ADL at the design phase and 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 our idea is similar to the interface mechanism in traditional CBSD (Component-Based Software Development), the basic concept of *Archface* is essentially different from that of CBSD. The typical interface mechanism does not expose anything without method signatures. As a consequence, it is difficult to encapsulate the collaboration among components into the interface definitions. The main purpose of the traditional interface mechanism is to provide a contract between client and provider components. Although the notion of DbC (Design by Contract) is effective for declaring a contract between them, it is necessary to analyse the whole of a program and trace the sequence of the method invocations between components in order to understand their collaboration. In *Archface*, a component interface exposes program points containing important architectural information such as method call, method execution, and control flow. A connector interface defines how to coordinate exposed program points. A collaborative architecture can be encapsulated into a group of component and connector interfaces. In *Archface*, program points are common constructs appeared in both of design and code. Model elements in a design model can be synchronized with program point shadows in the code via *Archface*.

The *Archface* interface mechanism is named *predicate coordination* in which the pointcut & advice mechanism in AOP (Aspect-Oriented Programming) [14] is used as a mechanism for exposing program points (pointcut-based predicates) and coordinating them (trait-based advice). By manipulating the architectural information exposed from program points, we can define a large and complex architecture.

The remainder of this paper is structured as follows. In Section 2, we point out the issues concerning design and implementation. In Section 3, the concept of *Archface* is illustrated. In Section 4, an overview of *Archface for Java*, a Java-based programming language supporting *Archface*, is shown. In Section 5, the effectiveness of *Archface* is evaluated. In Section 6, related work is introduced. Lastly, concluding remarks are provided in Section 7.

2. MOTIVATION

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

2.1 Example of architectural design

We use the *Observer* pattern as an example. The *Observer* pattern, one of the GoF design patterns [8], 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 `Subject` and `Observer`.

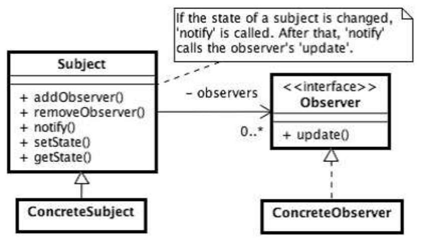


Figure 1: Observer pattern described in UML

When the state of a subject is changed, the subject notifies a new state to all observers.

Figure 1 illustrates the *Observer* pattern described in UML (Unified Modeling Language). In most cases, architectural design models are represented by using class diagrams, interaction diagrams, and state machine diagrams. Constraints are specified by OCL (Object Constraint Language). Design decisions that cannot be represented by neither diagrams nor OCLs are informally described in the form of notes by using natural languages. In Figure 1, the note shows that `notify` should be called under the control flow of `setState`.

Currently, most of the architectural design models are represented by diagrams as illustrated in Figure 1 because diagrams are institutive and easy to understand. However, it is not easy to check whether a design model contains defects because its diagram representation is not rigorous. Moreover, the diagram representation contains only restricted information. For example, only signature-based information (method signature) is contained in the class diagrams. A developer has to consider and guess the intent of architectural design when she or he translates diagrams into code. As a result, defects might be embedded into the code if the developer misunderstands the meanings of the design.

List 1 is a part of the `Subject` implementation.

```
[List 1]
public class Subject {
 private String state = "";
 public String getState() { return state; }
 public void setState(String s) { state = s; }
   ...
}
```

List 1 does not reflect the intention of the architectural design because `setState` only sets a new state and does not execute a notification task. Although many MDD tools can generate code from UML diagrams, most of them generate code such as List 1. A developer has to add extra code to the auto-generated code and might make a mistake. Embedded defects might be discovered by reviewing code with referring interaction diagrams. However, the review quality depends on the skill of the developers. 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 to architectural design. A design model should be at an adequately abstract level, not the same level as code.

Next, we discuss on the co-evolution between architectural design and program code. Currently, many MDD tools support round-trip engineering in which the code auto-generated from a design model contains modifiable regions specified by comments. These tools do not override the existing code in the modifiable regions when they re-generate the code from a revised design model. Both design and code can co-evolve if the code is modified within the regions. However, when a developer wants to modify the code that affects the architecture such as collaboration among components, it is difficult to reflect the code change on the design model because the round-trip engineering tools supporting the modifiable regions cannot recover the design model from the code but only re-generate the code from the design model. There remain difficult problems even if these tools can recover the design model from the code. A crucial problem is that all of the code changes should not always be reflected on the design model. For example, assume that `notify` is called in `setState`. This is an implementation faithful to its architectural design. Next, assume that a developer changes the old code to the new one in which `notify` is not called directly but a method is called from `setState` and the method calls `notify`. The round-trip engineering tools that can recover the design model from the code only reflect this calling sequence on the design model using an interaction diagram. However, this kind of design recovery is not sufficient for co-evolution because we cannot obtain a design model at the adequately abstract level. In this case, the design model has only constraints such that `notify` is called under the control flow of `setState`. So, the design model should not be changed even if the code is modified. A design model can be related to multiple code implementations. The MDD tools supporting simple round-trip engineering cannot deal with this relation between design and code.

2.2 Problems to be tackled

Problems between architectural design and implementation can be categorized into the following three items.

- *Abstraction level of architectural design*: It is not clear which abstraction level is adequate for designing architecture. The gap between design and implementation becomes large if the design is too abstract. On the other hand, the difference between them becomes unclear if the abstraction level of the design is low. A developer should design a software structure and clarify its rationale. However, it is not easy to find the proper abstraction level that can provide the traceability between design and implementation.

- *Refinement from design to code*: It is not easy to reflect the design decisions on the code when UML or informal natural languages are used for design notation. Although refinement using formal specification languages such as VDM, Z, and B is effective, it is not necessarily easy for ordinary developers to formally prove the correctness of the refinement. A method easy to introduce is favorable for these developers.

- *Co-evolution between design and code*: It is not easy to synchronize design and code. As mentioned before, current MDD approaches cannot necessarily help this co-evolution. Code should be modified if the design is changed. On the contrary, the design should be changed if the code is modified. The bidirectional traceability between design and code is needed.

These problems arise from a more essential cause: the boundary between design and implementation is vague. That is, it is not clear what kind of information we should specify in a design model and what kind of information we should program in the code. We have to explore a contract specification method between design and code.

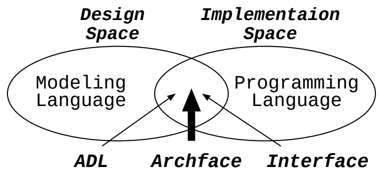


Figure 2: Archface

3. ARCHFACE

In this section, we illustrate the notion of *Archface* and show an example of an architecture description.

3.1 Basic concept

Figure 2 illustrates the role of *Archface* that resides in the intersection between design space and implementation space. *Archface* plays a role as ADL in the design space. At the same time, *Archface* plays a role of a language-level interface in the implementation space. In order to achieve this objective, we have to examine constructs appeared in both of modeling and programming languages. *Archface* provides a mechanism for exposing program points and coordinating them as a construct resided in both ADL and interface.

Three-part modeling framework

The *Archface* computation model is based on the three-part modeling framework [18] that explains a common structure in different AOP mechanisms such as AspectJ [15] and Hyper/J [12]. As shown in Figure 3, the framework explains each join point mechanism (JPM) in AOP as an interpreter which is modeled as a tuple of nine parameters:

$$\langle X, X_{JP}, A, A_{ID}, A_{EFF}, B, B_{ID}, B_{EFF}, META\rangle.$$

These parameters are divided into the following three parts: $A/A_{ID}/A_{EFF}$, $B/B_{ID}/B_{EFF}$, and $X/X_{JP}/META$. A and B are the languages in which the respective programs p_A and p_B, i.e., input to the interpreter, are written. X is the result domain of the weaving process, which is the third language of a computation. X_{JP} is the join point in X. A_{ID} and B_{ID} are the means, in the language A and B, of identifying elements of X_{JP}. A_{EFF} and B_{EFF} are the means, in the language A and B, of effecting semantics at the identified join points. $META$ is an optional meta-language for parameterizing the weaving process. A weaving process is defined as a procedure that accepts p_A, p_B, and $META$, and produces either a computation or a new program. This framework defines the process of AO weaving as taking two programs and coordinating them into a single combined computation. This framework describes the join points as existing in the result of the weaving process rather than residing in either of the input programs.

Mapping to Archface

In *Archface*, the three-part modeling framework is applied to represent the *component-and-connector* architecture. Although the original framework is proposed for modeling AOP, it can be applied not only to AOP but also to traditional

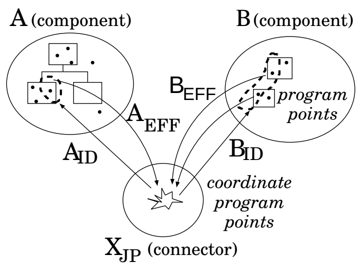


Figure 3: Component-and-connector architecture based on three-part modeling framework

CBSD. A and B are mapped to the *Archface* language construct for specifying components. A_{ID} and B_{ID} are mapped to pointcuts for exposing component's program points appeared in both of design and code. A_{EFF} and B_{EFF} are mapped to the advice for effecting the exposed program points. X is mapped to a trait-based connection for coordinating these program points. *Traits* [21] is a group of the pure methods that serve as a building block for classes. In *Archface*, coordination is described as a pure method that only handles the exposed program points without depending on the component implementation. A_{EFF}, B_{EFF}, and X are provided as a language construct for specifying connectors. X_{JP} is an event computed by a connector combining two component's program points. This component connection is performed by weaving. The component-binding via method calls is performed by weaving because method calls can be dealt with as a special case of weaving that coordinates a caller's *call* program point and a callee's *execution* program point.

Predicate coordination

In *Archface*, pointcut & advice in AOP is used as a mechanism for exposing program points (pointcut) and coordinating them (advice). Allen, R. and Garlan, D. proposed *Wright* [3], an ADL, to formalize the *component-and-connector* architecture. In *Write*, connectors are specified as a collection of protocols that characterize roles in an interaction. In *Archface*, connections are described as user-defined protocols that specify coordination among exposed program points.

Main characteristics of AOP can be explained with *predicate dispatch* [19] and *open class*. In *predicate dispatch*, each method can have a predicate guard specifying the conditions under which the method is invoked. *Open class* is a mechanism for adding methods/fields from the outside of a class. Inter-type declaration in AspectJ is a kind of *open class*.

We can characterize the notion of *Archface* with a simple word *"predicate coordination"* in which program points are exposed by a predicate (pointcut) and coordinated each other by a trait-based connector (advice). *Predicate coordination* includes the notion of *open class* because method/field can be dealt with as static program points. Predicates in *Archface* can be considered as a lightweight reflective mechanism for introspecting important architectural information such as program behavior and structures. By manipulating the information exposed from program points, we can define a large and rich architecture.

Below, we explain the concept of *Archface* more concretely by using code snippets written in *Archface for Java*. The detailed syntax is explained in Section 4.

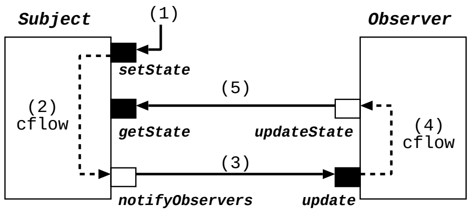


Figure 4: Collaborative architecture

3.2 Interface as ADL

Archface consists of two kinds of interfaces: *component* and *connector*.

List 2 is an *Archface* definition for the *Observer* patten shown in Figure 4. Here, `addObserver` and `removeObserver` in Figure 1 are omitted because of the space limitation. Only the notification interaction is described in List 2. *Archface* as an ADL specifies an architecture between *subject* and *observer* and does not contain implementation details.

```
[List 2]
01: interface component cSubject {
02:  pointcut getState():execution(String getState());
03:  pointcut setState():execution(void setState(String));
04:  pointcut notify():execution(void notify());
05:  pointcut notifyObservers() :
06:    cflow(execution(void setState(String)))
07:      && call(void notify());
08:
09:  port out getState();
10:  port out setState();
11:  port out notify();
12:  port in around() void :notifyObservers();
13: }
14:
15: interface component cObserver {
16:  pointcut update():execution(void update());
17:  pointcut updateState():
18:    cflow(execution(void update()))
19:      && call(String getSubjectState());
20:  port out update();
21:  port in around() String :updateState();
22: }
23:
24: interface connector cObserverPattern {
25:  connects notifyChange
26:  (port1 :cSubject.notifyObservers, port2 :cObserver.update){
27:     around() void :port1 { port2.proceed(); }
28:  }
29:  connects obtainNewState
30:  (port1 :cObserver.updateState, port2 :cSubject.getState){
31:     around() String :port1 { return port2.proceed(); }
32:  }
33: }
```

In *Archface*, a collaborative architecture is specified by a set of *ports* based on the pointcut mechanism. There are two kinds of ports: *in* for importation and *out* for exportation. By connecting *in* and *out* ports, program points can be exported or imported. In *Archface*, we can specify AspectJ pointcuts including `call` (method call), `execution` (method execution), and `cflow` (control flow). For example, the `notifyObservers` port (line 05-07, 12) exposes a `notify`'s `call` program point having such a constraint that `notify` has to be called under the control flow of `setState`. The operator && means *Logical AND*. This constraint is enforced when the `cSubject` interface is implemented. That is, a class implementing `cSubject` must satisfy the followings: 1) three public methods `getState`, `setState`, and `notify` must be defined; and 2) `notify` must be called under the control flow satisfying the above constraint.

As shown in Figure 4, we can understand the following from List 2 : 1) the state of `cSubject` is updated by calling `setState`; 2, 3) `update` in `cObserver` is called under the control flow of `setState` by connecting `notifyObservers` and `update` ports (line 05-07, 12, 20, 25-28); 4, 5) `getState` in `cSubject` is called under the control flow of `update` in `cObserver` (line 09, 17-19, 21, 29-32).

The `connects` statement coordinates program points exposed from ports. In the `notifyChange` statement, `around` advice connects a `notify`'s `call` program point to not its `execution` program point but an `update`'s `execution` program point. The `proceed` method performs a program point. In this case, the `port2.proceed` method performs the `update`'s `execution` program point. Conforming to the three-part modeling framework, an architectural event *"notify & update"* is computed by mediating `notify`'s *call* program point and `update`'s *execution* program point.

We think that *Archface* conceptually includes the notion of traditional interface mechanisms because a method signature can be related to the `execution` pointcut. The exposure of a method signature can be considered the exposure of a *method execution* program point.

Archface as ADL can be composed and verified at the architectural design level without considering implementation details (see 4.3 and 4.4).

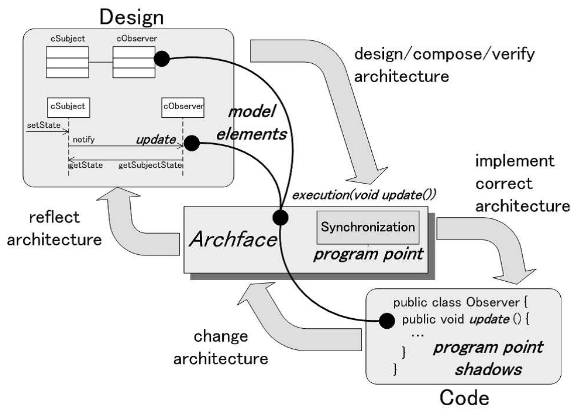
3.3 How to implement Archface

Although *Archface* enforces constraints on the class implementation, there can be a variety of actual implementation. For example, in List 2, it is not constrained which method calls `notify`. This is not specified in `cSubject`, and can be implemented in any ways. For example, calling sequences such as "`setState` → `mA` → `notify`" and "`setState` → `mA` → `mB` → `notify`" are possible.

List 3 is an implementation example of `cSubject` and `cObserver`. A developer has to implement the code that provides the program points exposed by *Archface* at the execution time. More concretely, the developer has to implement *program point shadows* that expose the program points when the program is executed.

Archface in List 2 is more abstract than the implementation in List 3 because the number of the former program points is less than that of the latter program points. For example, the `call` program point of `println` (line 17-18) is not contained in the *Archface* definitions (List 2). In *Archface*, only the essential part of the architectural design can synchronize with the code by using a set of program points selected by pointcut-based predicates. The `println` statement is not the target of this synchronization. The adequately abstract level can be kept between design and code.

```
[List 3]
01: architecture aObserverPattern {
02:  class Subject  implements cSubject;
03:  class Observer implements cObserver;
04: }
05:
06: public class Subject {
07:  private String state = "";
08:  public String getState() { return state; }
09:  public void setState(String s) { state = s; notify(); }
10:  void notify(){}
11: }
12:
13: public class Observer {
14:  private String state = "";
15:  public void update() {
16:    state = getSubjectState();
17:    System.out.println("Update received from Subject,
18:      state changed to : " + state);
19:  }
20:  String getSubjectState(){ return ""; }
21: }
```



```
public class Observer {
  public void update () {
    ...
  }
}
```

Figure 5: Software process with Archface

UML	Model element	Pointcut
Class diagram	class definition method definition field definition	class method field
Interaction diagram	message send message receive message sequence	call execution cflow

Table 1: Mapping from UML to Archface

3.4 Modeling and Archface

Figure 5 illustrates an *Archface*-centric software development process: 1) a modeler designs an architecture using a modeling tool that can manage the model in the form of *Archface*; 2) the modeler verifies the correctness and consistency of the design model; 3) a programmer develops the code conforming to the *Archface*; 4) if the architecture is changed at the programming phase, the corresponding *Archface* should be modified; and 5) the modeling tool can edit the revised architecture because the design model is managed in the form of *Archface*. Design and code can co-evolve each other with *Archface* that exists at the center of the development process. Model elements in the architectural design are bridged with program point shadows in the code via the program points specified in *Archface*.

Table 1 shows a program point mapping from UML model elements to *Archface* pointcuts. Although *Archface* does not depend on a specific modeling language, we use UML as an example. 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. Program points in the class diagrams are captured by structural pointcuts including `class/method/field` (explained in 4.1). Events in the interaction diagrams are mapped to method call/execution program points. Constraints related to control flow are mapped to `cflow`. If more rich pointcuts such as *association aspects* [20], data flow [18], and trace-match [2] are available, more rich architectural information can be encapsulated into an interface. Especially, *association aspects* is important to support multiplicity of the relationship among components. Although supporting these rich pointcuts is our future work, it is not clear what kinds of pointcuts are really needed for effective architectural descriptions. We think that it is preferable for a developer to be able to define domain-specific pointcuts.

Although *Archface* is conceptually language independent, it is preferable to generate a language dependent *Archface* (LDA) such as *Archface for Java* from a language independent *Archface* (LIA). As a future work, we plan to take an approach similar to IDL (Interface Description Language) compiler. This approach is also similar to the relation between PIM (Platform Independent Model) and PSM (Platform Dependent Model) in OMG's MDA (Model-Driven Architecture). LIA and LDA correspond to PIM and PSM, respectively. The syntax of LIA will be defined based on *Archface for Java* because the essential part of it is language independent although its surface syntax depends on Java.

3.5 How the problems are resolved

We pointed out three problems concerning design and implementation in Section 2.

The first problem is resolved by describing architecture in *Archface* because the essence of the *Observer* pattern can be described at the adequately abstract level without considering implementation details as shown in List 2.

The second problem is relaxed by implementing the code conforming to *Archface*. The *Archface* compiler detects an error if the code does not implement the *Archface* correctly.

The third problem is relaxed by the co-evolution between design and code. If the design model of the *Observer* pattern in List 2 is changed, List 3 might violate a new *Archface* reflecting the new design. In such a case, a programmer has to modify the code to conform to the new *Archface*. On the other hand, the design model should be changed when the code is changed with affecting *Archface*. As illustrated in Figure 5, model elements can be synchronized with program point shadows via *Archface*. Although traditional MDD tools use modifiable regions for supporting this kind of round-trip engineering, a tool supporting *Archface* does not need these regions because the round-trip can be performed via the program points appeared in both of design and code with keeping the adequately abstract level.

4. ARCHFACE FOR JAVA

Archface for Java is designed based on an AOP language *ccJava* [27] supporting the *component-and-connector* architecture. This language is previously proposed by us. After developing *ccJava*, we found that it could be generalized to support not only AOP but also traditional CBSD. Moreover, we found that our idea could be applied to not only programming but also architectural design. In this paper, we purified our idea as *predicate coordination*. In *Archface for Java*, *ccJava* language features specific to AOP are removed and more general language constructs are introduced.

In this section, we show an overview of *Archface for Java*.

4.1 Language features

Table 2 shows main language constructs.

Component and connector interface

A *component* interface consists of port and pointcut declarations. Although the syntax of pointcut designators is basically the same as that of AspectJ, there is a crucial difference: *Archface* includes no class specification in the signatures. For example, `execution(String getState())` is not described as `Subject.getState()`. This separates the `cSubject` interface (List 2: line 01-13) from the `Subject` class (List 3: line 06-11). A programmer that uses the `cSubject` interface does not have to be aware of the existence of the `Subject` class. Component implementation is specified by the `architecture` statement (List 3: line 01-04).

Interface	Feature	Reserved word
Component Interface	port	in, out
	pointcut	class, method, field (static)
		call, execution, cflow, etc. (dynamic)
	inheritance	extends
	implementation	architecture, implements
Connector Interface	connection	connects
	advice	introduce (static)
		before, after, around (dynamic)

Table 2: Archface for Java

A *connector* interface represents connections among ports. The types of advice that can be applied to an in-port are declared in an `in` statement. In List 2 (line 12), only `around` advice can be applied to the `notifyObservers` port.

Pointcut and advice

Pointcuts in *Archface* are based on AspectJ. The targets of importing or exporting program points are not restricted to `call/execution/cflow`: all of the primitive pointcut designators in AspectJ can be used. Furthermore, *Archface* provides additional pointcut designators such as `class`, `method`, and `field`. `Class/method/field` select a set of class, method, field definitions. These definitions are considered static program points. Program points such as method execution are considered dynamic. Static pointcuts and dynamic pointcuts cannot be used in the same port definition.

All of the advice types including `before`, `after`, and `around` in AspectJ can be used in *Archface*. We can use additional advice `introduce` that introduces imported methods or fields to an original class. The `introduce` advice applied to only static program points corresponds to *open class*.

A developer can design not only behavioral but also structural architecture by using static program points (see 4.3).

Abstract Archface

Archface descriptions reused in many applications should not depend on a specific application. Although List 2 well represents the *Observer* pattern, this *Archface* depends on specific component signatures such as `setState(String)`. However, all of the applications that can apply the *Observer* pattern do not always manage a state as a string.

To deal with this problem, *Archface* provides the abstract interface mechanism for specifying only the design outline. List 4 shows an abstract *Archface* `aSubject` and its concrete *Archface* `cSubject` defining concrete pointcuts. Although the former does not contain concrete method signatures. it represents the architectural essence in the *Observer* pattern. The subtyping in *Archface* can be explained by the subset relation of exposed program points between super and sub interfaces.

```
[List 4]
interface component aSubject {
 pointcut getState();
 pointcut setState();
 pointcut notify();
 pointcut notifyObservers() : cflow(setState()) && notify();
   ...
}

interface component cSubject extends aSubject {
 pointcut getState(): execution(String getState());
 pointcut setState(): execution(void setState(String));
 pointcut notify(): call(void notify());
}
```

4.2 AO architecture

Archface supports not only traditional CBSD but also AO architecture. We do not have to distinguish crosscutting concerns from primary concerns.

List 5 shows a graphical editor program consisting of `cPoint`, `cLine`, and `cDisplay`. The `cDisplayUpdate` connector signals `cDisplay` to update shapes whenever their coordinates are changed. The operator || means *Logical OR*.

A developer can easily understand which component crosscuts over other components. In List 5, `cDisplay` crosscuts over `cPoint` and `cLine` (line 22-23). The decision of *crosscutting or primary* only depends on whether a port is connected to multiple ports or not.

```
[List 5]
01: interface component cPoint {
02:  pointcut change():
03:   execution(void setX(int)) || execution(void setY(int)) ||
04:   execution(void moveBy(int, int));
05:  port in before(), after() returning, around() : change();
06: }
07:
08: interface component cLine {
09:  pointcut change():
10:   execution(void setP*(Point)) ||
11:   execution(void moveBy(int, int));
12:  port in before(), after() returning, around() : change();
13: }
14:
15: interface component cDisplay {
16:  pointcut redraw(): execution(void update());
17:  port in before(), after() returning : redraw();
18:  port out redraw();
19: }
20:
21: interface connector cDisplayUpdate {
22:  connect(port1 :cDisplay.redraw,
23:          port2 :{cPoint || cLine}.change){
24:     after() returning : port2 { port1.proceed(); }
25:  }
26: }
```

In *Archface*, wildcard can be used in the pointcut definitions (line 10). For example, `setP*` can be matched with `setP1` and `setP2`. Using abstract *Archface* and wildcard, reusable crosscutting components can be easily defined.

4.3 Multi-view architectural composition

Since software architecture consists of multiple views including structural and behavioral aspects, it is preferable to describe these views independently in terms of MDSOC (Multi-Dimensional Separation Of Concerns) [25].

Archface can represent MDSOC based on Hyper/J-like architecture as shown in List 6.

```
[List 6]
interface component cColorView {
  pointcut color_property() :
   field(int color) ||
   method(void setColor(int))||
   method(int getColor());
  port out color_property();

interface component cPointView extends cPoint {
  pointcut thisClass() : class(this);
  port in introduce() : thisClass();
}

interface connector cMerge {
  connect(port1:cColorView.color_property,
          port2:cPointView.thisClass){
   introduce() : port2 from port1;
}
```

The `color` field and associated getter/setter methods are introduced to a point. List 6 is an example of *open class* using static program points. Two viewpoints `cColorView` and `cPointView` can be composed together by the `cMerge` connector. Using MDSOC-type connections, we can design a layered architecture. List 6 can be considered as a layer structure consisting of point and color layers.

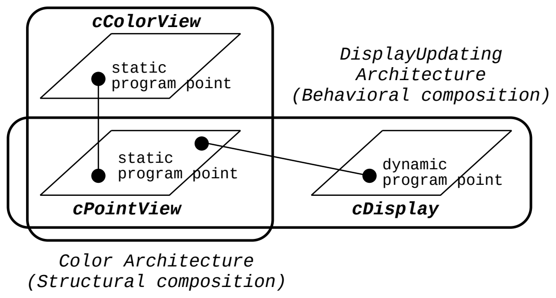

cColorView
static program point
DisplayUpdating Architecture (Behavioral composition)
static program point
dynamic program point
cPointView
cDisplay
Color Architecture (Structural composition)

Figure 6: Architectural composition

By defining connectors, a set of small architectural views can be composed together to construct a large architecture. In *Archface*, special kind of connectors are not needed to bridge two architectures. Ordinary connectors can be used. A connector coordinates program points exposed by one architecture and program points exposed by another one. In this case, the precedence of connectors might affect the semantics of a composed architecture. In *Archface*, precedence is specified by the appearance order of connectors. As illustrated in Figure 6, *Display Updating* architecture in List 5 and *Color* architecture in List 6 can be composed together. The former represents a dynamic aspect of architectural design and the latter represents a static aspect. `cColorView` is composed with `cPointView` in order to update a color when a position of a point is changed. In some situation, a developer might has to convert program points exposed by one port to other kinds of program points that can be handled by another port because names or signatures are differently used between two architectures. In this case, the developer has to define filter connectors to bridge the gap between two architectures. `cMerge` in List 6 can be considered a kind of filter that converts a port exposing program points *"execution of position change"* to another port exposing program points consisting of *"execution of position and color change"*.

4.4 Architectural verification

It is preferable to check the correctness and consistency of software architecture at the design level.

We show a verification method using model checking that can check the reachability of control flow and behavioral conflicts. If architecture is verified at the design phase, we do not have to verify the architectural correctness at the programming phase. In the modeling phase, the consistency and correctness of architectural design specified by *Archface* is verified. In the implementation phase, the *Archface* compiler and associated testing tools check whether a program conforms to the *Archface* definitions.

Here, we use SPIN [11] as a model checker. List 7 is the PROMELA code translated from *Archface* descriptions in List 2. Components in *Archface* are mapped to processes in PROMELA (line 6, 11). Connectors are also mapped to processes (line 16, 21). Port *in* and *out* are mapped to message receive (?) and send (!), respectively (line 8, 13, 18, 23). Control flow is translated into the simplest case (line 8, 13) because the purpose of the verification is not to verify program code but to check the inconsistency of architecture descriptions. So, we only have to check the simplest case.

```
[List 7]
#define SYNCH 0
mtype = {setState, getState, notify, update, getSubjectState}
chan toSubject = [SYNCH] of {mtype};
chan toObserver = [SYNCH] of {mtype};

proctype cSubject(){
  do
    :: to_Subject?setState -> to_Subject!notify
  od
}
proctype cObserver(){
  do
    :: to_Observer?update -> to_Observer!getSubjectState
  od
}
proctype notifyChange(){
  do
    :: toSubject?notify -> toObserver!update
  od
}
proctype ObtainNewState(){
  do
    :: toObserver?getSubjectState -> toSubject!getState
  od
}
```

We can verify whether the *Archface* in List 2 satisfies the following constraint: if `setState` in `cSubject` is called, `getState` is called from `cObserver`. This constraint can be specified by an LTL (Linear Temporal Logic) formula.

4.5 Prototype implementation

The *Archface* compiler translates *Archface* definitions into an AspectJ program. In the generated code, components and connections are represented as a set of aspects and classes. First, from a connector interface, the compiler generates connective relations among component interfaces and checks if a specified class implements a corresponding component interface. Next, a component interface is converted to an aspect definition in AspectJ.

List 8 is the code generated from List 2 (`cSubject` and `cObserver` components, `cObserverPattern` connector) and List 3 (`Subject` and `Observer` classes). The contents of the connector interface is converted to the advice in the generated aspects (line 11, 22). For example, `notifyChange` (List 2: line 25-28) is translated into the `around` advice in the `cSubject` aspect. Factory classes (line 10, 21) are needed to generate an executable program reflecting the collaboration specified by *Archface*. Although default factory classes are automatically generated, a developer can specify user-defined factory classes.

```
[List 8]
01: aspect cSubject {
02:  pointcut getState():
03:    execution(String getState()) && within(Subject);
04:  pointcut setState():
05:    execution(void setState(String)) && within(Subject);
06:  pointcut notifyObservers():
07:    cflow(execution(void setState(String)))
08:    && call(void notify()) && within(Subject);
09:
10:  Observer m_observer = ObserverFactory.getInstance();
11:  void around(): notifyObservers(){m_observer.update();}
12: }
13:
14: aspect cObserver {
15:  pointcut update():
16:    execution(void update()) && within(Observer);
17:  pointcut updateState():
18:    cflow(execution(void update()))
19:    && call(String getSubjectState()) && within(Observer);
20:
21:  Subject m_subject = SubjectFactory.getInstance();
22:  String around(): updateState(){return m_subject.getState();}
23: }
```

The compiler detects an error if the implementation in List 3 does not conform to the *Archface* in List 2. If `Observer` does not contain the `update` method, the compiler displays a message "the method *update* is undefined for the type Observer". Although most of the inconsistencies between design and code can be checked by the compiler, some kinds of defects are not always detected by only static analysis because contextual pointcuts include dynamic properties. Inconsistency checking should integrate static analysis with dynamic analysis and testing. For testing, we can use aspects that monitor the execution sequence of program points. List 9 is an aspect for checking whether `getState` is executed after `setState` is executed.

```
[List 9]
aspect Test {
 pointcut testCallingSequence():
   cflow(execution(void Subject.setState(String)))
   && execute(String Subject.getState());
 after() : testCallingSequence(){
   System.out.println("OK!");
}
```

In our current implementation, the consistency check is preformed by only type checker. We plan to support a testing facility mentioned above. Furthermore, we plan to develop a modeling tool for editing restricted UML diagrams that can be mapped to *Archface* (see 3.4) and verifying architecture (see 4.4). After developing these tools, we can automate the whole of the software development process illustrated in Figure 5. At this time, we have developed the *Archface* language processor, the heart of Figure 5.

POSA	Archface Component	Evaluation
Layer	Each layer	Well-fitted
Pipes&Filters	Data Source, Filter, Data Sink	Well-fitted
Blackboard	Blackboard, Knowledge Source, Control	Well-fitted
Broker	Client, Client-side Proxy, Broker, Server-side Proxy, Server	Well-fitted
MVC	Model, View, Controller	Well-fitted
PAC	{Top,Interm.,Bottom}-Level Agent	Well-fitted
Micro kernel	Internal server, External server, Adapter, Client, Microkernel	Fair
Reflection	{Base,Meta}-Level, Metaobject Protocol	Fair

MVC (Model-View-Controller)
PAC (Presentation-Abstraction-Control)

Table 3: POSA patterns and Archface

5. DISCUSSION

We evaluate the architectural expressiveness by using the POSA patterns as the criteria for discussing how *Archface* can represent well-known architecture. Next, we compare the difference between *Archface* and other traditional ADLs.

5.1 Architectural expressiveness

Table 3 shows the relation between the POSA patterns and *Archface*. The pattern catalogues in POSA consist of example, context, problem, solution, structure, dynamics, implementation, variants, known uses, example resolved, consequences, and credits. Especially, structure and dynamics are important for *Archface* descriptions because components and connectors can be extracted from these catalogue items.

Most of the POSA patterns are considered the *component-and-connector* architecture and can be translated into *Archface* descriptions. Although the component granularity of the POSA patterns is relatively large, these architectures can be described by using abstract *Archface*. Architectural patterns except *Micro kernel* and *Reflection* are well-fitted to *Archface*. However, these two patterns are not enough represented by *Archface* because the role of the components appeared in these patterns are not clear. For example, the

Microkernel pattern consists of `External Server`, `Microkernel`, and `Internal Server`. Although these components can be represented by *Archface*, it is difficult to describe `Microkernel` as one component in real world applications.

NFR (Non Functional Requirements) such as performance and security are important to design high quality architecture. If NFR can be encapsulated as crosscutting components, NFR can be represented by *Archface*.

Here, we summarize the result of our evaluation. If component roles are clear and NFR can be modularized as a crosscutting component, the architecture can be well represented by *Archface*. Most of the POSA patterns are included in this category. If component roles are not explicit, *Archface* is not suitable for architectural descriptions. Although these patterns provide design strategies or abstract design outlines, the patterns are not directly related to the concrete architectural design. For example, the `Microkernel` component in the *Micro kernel* pattern does not show how to design a micro kernel itself. We think that an architecture based on *Micro kernel* or *Reflection* should be divided into a set of more concrete architectural models that can be represented by *Archface*.

5.2 Comparison with traditional ADLs

There are many *component-and-connector*-based ADLs including *Darwin* [16] and *Wright* [3]. In *Darwin*, FSP (Finite State Process) is used to describe behavior and verify it. In *Wright*, dynamic behavior is described using CSP (Communicating Sequential Processes). These ADLs enable a developer to describe software architecture rigorously and verify it. Moreover, some ADL support tools can generate code from an architecture. Allen, R. and Garlan, D. formalized architectural connection and provided a verification method based on model checking [3]. What is a difference between *Archface* and these traditional ADL's approaches ?

We can point out following three differences: 1) traditional ADLs cannot support language-level bidirectional traceability; 2) richness of the architectural expressiveness only exists in ADLs, and 3) type of the architecture is fixed and cannot be extended. First, we think about 1). Although there are ADLs that can generate program code, it is difficult to recover architectural design from the generated code because the architecture is embedded into the code. On the other hand, in *Archface*, an architecture is encapsulated into the interfaces. We can understand the architecture at the code-level. The architectural design also can be recovered from these interfaces. As a result, bidirectional traceability can be realized in *Archface*. Next, we think about the second difference. This difference is related to the first one. In traditional ADL's approaches, architectural information does not exist in program code but only in ADL' descriptions. Due to this asymmetricity, it is difficult for a developer to co-evolve design and implementation. This problem is relaxed in *Archface* as claimed in this paper. Lastly, we discuss on the third difference. In traditional ADLs, we cannot extend the ability of the architecture expressiveness. For example, an ADL supporting only control-flow-based component interactions cannot support data flow without redesigning and reimplementing the ADL. In *Archface*, the key idea is *predicate coordination* in which a predicate is a kind of reflective mechanism for introspecting program behavior and structures. By defining new predicates, we can extend the architectural expressiveness as discussed in 3.4.

6. RELATED WORK

Aldrich, J. et al. proposed ArchJava [1], an extension to Java. ArchJava unifies architecture with implementation, ensuring that the implementation conforms to architectural constraints. *Archface* enhances this approach and separates architecture definitions from actual implementation by introducing a new interface mechanism. ObjectTeams [10] can describe collaboration as a module by introducing two kind of modules *teams* and *roles*. Although ObjectTeams is similar to *Archface*, the collaboration in ObjectTeams is not a design-level construct but a programming-level construct.

Larch [28], a language for formally specifying program modules, uses two tiers. The top tier is a behavioral interface specification language (BISL) that uses pre-/post-conditions. BISL is tailored to a specific programming language. The bottom tier is the Larch Shared Language (LSL) for describing the mathematical vocabulary used in the pre-/post-conditions. Although the target of *Larch* is different from that of *Archface*, both language structures are slightly similar. BISL and LSL can be mapped to *Archface* language and its pointcut definition language. *Archface* can be tailored to a specific language such as Java.

There are some works that adopt pointcuts as an interface mechanism. *Crosscut programming interface* (XPI) [23] is an interface for specifying design rules between aspects and classes. In *Archface*, pointcuts are used as an interface between architectural design and implementation.

Using a set of abstract classes, an architecture can be described by the collaboration among abstract classes. Design patterns such as the *Observer* pattern can be represented in this way. Hannemann, J. and Kiczales, G. provided design pattern implementations in Java and AspectJ [9]. However, in these cases, implementation is included in abstract classes or aspects. On the other hand, *Archface* is a type that enforces the architectural design on its implementation. In the traditional framework approaches, we have to read code when we want to understand its design at the programming phase. In *Archface*, we can understand the design by only reading *Archface* that is bridged with a design model.

Our approach is closely related to the research on co-evolution between design and implementation. D'Hondt, T. et al. introduced the logic-meta programming (LMP) in order to enforce the synchronization between OO design and code [7]. Although *Archface* is similar to their approach in which software design is expressed as logic meta programs over the implementation, our approach is based on the interface mechanisms that not only enforce architectural constraints on the program implementation but also represent architectural abstractions. Cazzola, W. et al. proposed an approach that deals with the problem of co-evolving the application design models after the code refactoring [6]. In their approach, a developer decorates the code with meta-data describing how its design should be adapted after the developer changes the application code. In *Archface*, a developer does not have to add extra meta-data for synchronizing between design and code because the *Archface* compiler checks whether modified new code conforms to its *Archface*. If there are code modifications that affect its design, the code does not conform to its *Archface*. In this case, the developer has to modify the *Archface* descriptions. The idea of adopting such a co-evolution approach to deal with AOP at a higher-level of abstraction than just source code has also been proposed by several researchers [13].

Aspectual components can be developed by adopting *Archface* in which AO can be introduced only with the *component-and-connector* architecture. JAsCo [24] is an AO implementation language that introduces aspect beans and connectors. An aspect bean describes behavior that interferes with the execution of a component by using a hook. A connector is used for deploying hooks within a specific context. Although both of *Archface* and JAsCo are based on the *component-and-connector* mechanism, the roles of components and connectors in *Archface* are different from those in JAsCo. In *Archface*, we do not have to distinguish an aspect component from an ordinary component.

7. CONCLUSIONS

This paper proposed *Archface* that is not only an ADL but also a programming-level interface mechanism.

Shaw, M. and Garlan, D. identified six properties that an ADL should provide: composition, abstraction, reusability, configuration, heterogeneity, and analysis [22]. *Archface* satisfies these properties. Component interfaces can be composed together by a connector interface. Architecture can be abstracted using *Archface* that does not include implementation details. An interface defined in one architectural design model can be reused in other models. Configuration can be realized by connecting a set of component interfaces. *Archface* supports heterogeneity because several kinds of architecture types including OO-based collaboration, AO, and MDSOC can be represented based on *predicate coordination*. Analysis and verification can be performed by using a model checker.

Archface facilitates architectural reuse, composition, evolution, and verification as demonstrated in this paper. *Archface* provides a contract place where design and code meet together. Our main idea, *predicate coordination*, is very simple and we believe that the idea opens a new research direction discussing what is design and what is implementation.

8. 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] Allan, C., Avgustinov, P., Christensen, A.S., Hendren, L., Kuzins, S., Lhoták, O., de Moor, O., Sereni, D., Sittampalam, G., and Tibble, J.: Adding Trace Matching with Free Variables to AspectJ, In *Proceedings of the 20th annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2005)*, pp.345-364, 2005.

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

[4] Bass, L., Clements, P., and Kazman, R.: *Software Architecture in Practice (2nd edition)*, Addison-Wesley, 2003.

[5] Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., and Stal, M.: *Pattern-Oriented Software Architecture –A System of Patterns*, John Wiley & Sons, 1996.

[6] Cazzola, W., Pini, S., Ghoneim, A., and Saake, G.: Co-Evolving Application Code and Design Models by Exploiting Meta-Data, In *Proceedings of the 12th Annual ACM Symposium on Applied Computing (SAC 2007)*, pp.1275-1279, 2007.

[7] D'Hondt, T., Volder, K. D., Mens, K., and Wuyts, R., Co-evolution of Object-Oriented Software Design and Implementation, In *Aksit, M. (Ed.), Software Architectures and Component Technology (SACT 2000)*, Kluwer Academic Publishers, pp.207-224, 2001.

[8] Gamma, E., Helm, R., Johnson, R., and Vlissides, J. M.: *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1995.

[9] Hannemann, J. and Kiczales, G.: Design Pattern Implementation in Java and AspectJ, In *Proceedings of the 17th annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2002)*, pp.161-173, 2002.

[10] Herrmann, S.: A Precise Model for Contextual Roles: The Programming Language ObjectTeams/Java, *Applied Ontology*, vol. 2, no. 2, pp.181-207, IOS Press, 2007.

[11] Holzmann, G. J.: *The SPIN MODEL CHECKER*, Addison-Wesley Pub, 2003.

[12] Hyper/J, http://www.research.ibm.com/hyperspace/HyperJ/HyperJ.htm

[13] Kellens, A., Mens, K., Brichau, J., and Gybels, K.: Managing the Evolution of Aspect-Oriented Software with Model-based Pointcuts, In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP 2006)*, pp.501-525, 2006.

[14] 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.

[15] 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.

[16] Magee, J. and Kramer, J.: Dynamic Structure in Software Architectures, *ACM SIGSOFT Software Engineering Notes*, Volume 21, Issue 6, pp.3-14, 1996.

[17] Masuhara, H. and Kiczales, G.: Modeling Crosscutting in Aspect-Oriented Mechanisms, In *Proceedings of European Conference on Object-Oriented Programming (ECOOP 2003)*, pp.2-28, 2003.

[18] Masuhara, H. and Kawauchi, K.: Dataflow Pointcut in Aspect-Oriented Programming, In *Proceedings of The First Asian Symposium on Programming Languages and Systems (APLAS'03)*, pp.105-121, 2003.

[19] Millstein, T.: Practical Predicate Dispatch, In *Proceedings of the 19th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2004)*, pp.345-364, 2004.

[20] Sakurai, K., Masuhara, H., Ubayashi, N., Matsuura, S., and Komiya, S.: Association Aspects, In *Proceedings of the 3rd International Conference on Aspect-Oriented Software Development (AOSD 2004)*, pp.16-25, 2004.

[21] Schärli, N., Ducasse, S., Nierstrasz, O., and Black, A.: Traits: Composable Units of Behavior, In *Proceedings of European Conference on Object-Oriented Programming (ECOOP 2003)*, LNCS 2743 pp.248-274, 2003.

[22] Shaw, M. and Garlan, D.: Characteristics of Higher Level Languages for Software Architecture, *Technical Report, CMU-CS-94-210*, Carnegie Mellon University, 1994.

[23] Sullivan, K., Griswold, W., Song, Y., Cai, Y., Shonle, M., Tewari, N., and Rajan, H.: Information Hiding Interface for Aspect-Oriented Design, In *Proceedings of the 5th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE 2005)* , pp.166-175, 2005.

[24] Suvée, D., Vanderperren, W., and Jonckers, V.: JAsCo: An Aspect-Oriented Approach Tailored for Component Based Software Development, In *Proceedings of the 2nd International Conference on Aspect-Oriented Software Development (AOSD 2003)*, pp.21-29, 2003.

[25] Tarr, P., Ossher, H., Harrison, W., and Sutton, S.M., Jr.: N Degrees of Separation: Multi-dimensional Separation of Concerns, In *Proceedings of the 21st International Conference on Software Engineering (ICSE'99)*, pp.107-119, 1999.

[26] 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.

[27] Ubayashi, N., Sakai, A., and Tamai, T.: An Aspect-oriented Weaving Mechanism Based on Component and Connector Architecture, In *Proceedings of the 22nd IEEE/ACM International Conference on Automated Software Engineering (ASE 2007)*, pp.154-163, 2007.

[28] Wing, Jeannette M.: Writing Larch Interface Language Specifications, *ACM Transactions on Programming Languages and Systems*, 9(1), pp.1-24, 1987.