

# Component-Oriented Programming in ArchJava

Jonathan Aldrich

Craig Chambers

David Notkin

Department of Computer Science and Engineering  
University of Washington  
Box 352350  
Seattle, WA 98195-2350 USA  
+1 206 616-1846

{jonal, chambers, notkin}@cs.washington.edu

## Abstract

Component-oriented programming supports constructing software systems by composing independent components into a software architecture. However, existing approaches decouple implementation code from architecture, allowing inconsistencies, causing confusion, violating architectural properties, and inhibiting software evolution. ArchJava is an extension to Java that seamlessly unifies software architecture with implementation, ensuring that the implementation conforms to architectural constraints. A case study applying ArchJava to a circuit-design application suggests that ArchJava can express architectural structure effectively within an implementation, and that it can aid in program understanding and software evolution.

## 1. Introduction

Software architecture [GS93][PW92] is the organization of a software system as a collection of interacting components. A typical architecture includes a set of components, connections between the components, and constraints on how components interact. Describing architecture in a formal architecture description language (ADL) [MT00] can make designs more precise and subject to analysis, as well as aid program understanding, implementation, evolution, and reuse.

Existing ADLs, however, are loosely coupled to implementation languages, causing problems in the analysis, implementation, understanding, and evolution of software systems. Some ADLs [SDK+95][LV95] and CASE tools connect components that are implemented in a separate language. However, these languages do not guarantee that the implementation code obeys architectural constraints, but instead rely on developers to follow style guidelines that prohibit common programming idioms such as data sharing. Architectures described with more abstract ADLs [AG97][MQR95] must be implemented in an entirely different language, making it difficult to trace architectural features to the implementation, and allowing the implementation to become inconsistent with the architecture as the program evolves. Thus, analysis in existing ADLs may reveal important architectural properties, but these properties are not guaranteed to hold in the implementation.

In order to enable architectural reasoning about an implementation, the implementation must obey a consistency property called *communication integrity* [MQR95][LV95]. A system has communication integrity if implementation components only communicate directly with the components they are connected to in the architecture.

This paper presents ArchJava, a small, backwards-compatible, component-oriented extension to Java that integrates software architecture smoothly with Java implementation code. Our design makes two novel contributions:

- ArchJava seamlessly unifies architectural structure and implementation in one language, allowing flexible implementation techniques, ensuring traceability between architecture and code, and supporting the co-evolution of architecture and implementation.
- ArchJava also guarantees communication integrity in an architecture's implementation, even in the presence of advanced architectural features like run time component creation and connection.

To evaluate our approach, we applied ArchJava to Aphyds, a moderate-size circuit design application. Using an informal architecture diagram hand-drawn by the developer as our guide, we reengineered Aphyds to make this architecture explicit in the implementation code. The resulting architecture revealed inconsistency and complexity in the communication between components, and made it easier to refactor the program to clarify the communication. Our experience also suggests that it may be easier to understand and evolve the resulting program than the original program.

The rest of this paper is organized as follows. After the next section's discussion of related work, section 3 introduces the ArchJava language. Section 4 describes our experience applying ArchJava to Aphyds. Finally, we conclude with a discussion of future work.

## 2. Previous Work

A number of architecture description languages have been defined to describe, model, check, and implement software architectures [MT00]. Many ADLs support sophisticated analysis, such as checking for protocol deadlock [AG97] or formal reasoning about correct refinement [MQR95]. Some ADLs allow programmers to fill in implementation code to make a complete system [LV95][SDK+95]. However, there is no guarantee that the implementation respects the software architecture unless programmers adhere to certain style guidelines.

Tools such as Reflexion Models [MNS01] have been developed to show an engineer where an implementation is and is not consistent with an architectural view of a software system. These tools are particularly effective for legacy systems, where rewriting the application in a language that supports architecture directly would be prohibitively expensive.

A number of computer-aided software engineering tools allow programmers to define a software architecture in a design language such as UML, ROOM, or SDL, and fill in the architecture with code in the same language or in C++ or Java. While these tools have powerful capabilities, they either do not enforce communication integrity or enforce it in a restricted language that is only applicable to certain domains. For example, the SDL embedded system language prohibits all data sharing between components via object references. This restriction ensures communication integrity, but it also makes these languages very awkward for general-purpose programming. Many UML tools such as Rational Rose or I-Logix Rhapsody, in contrast, allow method implementations to be specified in a language like C++ or Java. This supports a great deal of flexibility, but since the C++ or Java code may communicate arbitrarily with other system components, there is no guarantee of communication integrity in the implementation code.

Component-based infrastructures such as COM, CORBA, and JavaBeans provide sophisticated services such as naming, transactions and distribution for component-based applications. Some commercial tools even provide graphical ways to connect components together, allowing simple architectures to be visualized. However, these systems have poor support for structural specification of dynamically changing systems, and have no concept of communication integrity. Communication integrity can only be enforced by programmer discipline following guidelines such as the Law of Demeter [LH89] that states, “only talk to your immediate friends” in a system.

Advanced module systems such as ML’s functors [MTH90] can be used to encapsulate components and to describe the static architecture of a system. The FoxNet project [B95] shows how functors can be used to build up a network stack architecture out of statically connected components. However, these systems do not guarantee communication integrity in the language; instead, programmers must follow a careful methodology to ensure that each module communicates only with the modules it is connected to in the architecture. ArchJava’s focus on the structure of component *instances* and communication integrity complements the goals of Jiazzi [MFH01], which provides a flexible way to link Java *classes* into modules and enforces module encapsulation.

More recently, the component-oriented programming languages ComponentJ [SC00] and ACOEL [Sre01] extend a Java-like base language to explicitly support component composition. These languages can be used to express components and static architectures. However, neither language makes dynamic architectures explicit, and neither enforces communication integrity.

### 3. The ArchJava Language

ArchJava is designed to investigate the benefits and drawbacks of a relatively unexplored part of the ADL design space. Our approach extends a practical implementation language to incorporate architectural features and enforce communication integrity. Key benefits we hope to realize with this approach include better program understanding, reliable architectural reasoning about code, keeping architecture and code consistent as they evolve, and encouraging more developers to take advantage of software architecture. ArchJava’s design also has some limitations, discussed below in section 3.6.

```
public component class Parser {
    public port in {
        provides void setInfo(Token symbol,
                               SymTabEntry e);
        requires Token nextToken()
                throws ScanException;
    }
    public port out {
        provides SymTabEntry getInfo(Token t);
        requires void compile(AST ast);
    }

    void parse(String file) {
        Token tok = in.nextToken();
        AST ast = parseFile(tok);
        out.compile(ast);
    }

    void parseFile(Token lookahead) { ... }
    void setInfo(Token t, SymTabEntry e) { ... }
    SymTabEntry getInfo(Token t) { ... }
    ...
}
```

**Figure 1. A parser component in ArchJava. The Parser component class uses two ports to communicate with other components in a compiler. The parser’s in port declares a required method that requests a token from the lexical analyzer, and a provided method that initializes tokens in the symbol table. The out port requires a method that compiles an AST to object code, and provides a method that looks up tokens in the symbol table.**

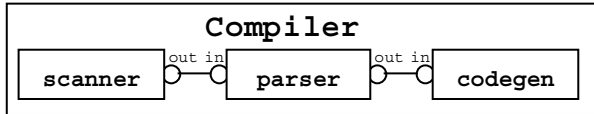
A prototype compiler for ArchJava is publicly available for download at the ArchJava web site [ACN01]. Although in ArchJava the source code is the canonical representation of the architecture, visual representations are also important for conveying architectural structure. This paper uses hand-drawn diagrams to communicate architecture; however, we have also constructed a simple visualization tool that generates architectural diagrams automatically from ArchJava source code. In addition, we intend to provide an archjavadoc tool that would automatically construct graphical and textual web-based documentation for ArchJava architectures.

To allow programmers to describe software architecture, ArchJava adds new language constructs to support *components*, *connections*, and *ports*. The rest of this section describes by example how to use these constructs to express software architectures. Throughout the discussion, we show how the constructs work together to enforce communication integrity, culminating in a precise definition of communication integrity in ArchJava. Reports on the ArchJava web site [ACN01] provide more information, including the complete language semantics and a formal proof of communication integrity in the core of ArchJava.

#### 3.1. Components and Ports

A *component* is a special kind of object that communicates with other components in a structured way. Components are instances of *component classes*, such as the Parser component class in Figure 1. Component classes can inherit from other components.

A component instance communicates with external components through ports. A *port* represents a logical communication channel between a component instance and one or more components that it is connected to.



```

public component class Compiler {
    component Scanner scanner;
    component Parser parser;
    component CodeGen codegen;

    connect scanner.out, parser.in;
    connect parser.out, codegen.in;

    public static void main(String args[]) {
        new Compiler().compile(args);
    }

    public void compile(String args[]) {
        // for each file in args do:
        ...parser.parse(file);...
    }
}

```

**Figure 2.** A graphical compiler architecture and its ArchJava representation. The `Compiler` component class contains three subcomponents—a `Scanner`, a `Parser`, and a `CodeGen`. This compiler architecture follows the well-known pipeline compiler design [GS93]. The `scanner`, `parser`, and `codegen` components are connected in a linear sequence, with the `out` port of one component connected to the `in` port of the next component.

Ports declare three sets of methods, specified using the `requires`, `provides`, and `broadcasts` keywords. *Provided* methods can be invoked by other components connected to the port. The component can invoke a disjoint set of *required* methods through the port. Each required method is implemented by a component that the port is connected to. *Broadcast* methods are just like required methods, except that they must return `void` and may be connected to an unbounded number of implementations.

A port specifies both the services implemented by a component and the services a component needs to do its job. Required interfaces make dependencies explicit, reducing coupling between components and promoting understanding of components in isolation. Ports also make it easier to reason about a component’s communication patterns.

Each port is a first-class object that implements its required and broadcast methods, so a component can invoke these methods directly on its ports. For example, the `parse` method calls `nextToken` on the `parser`’s `in` port. These calls will be bound to external components that implement the appropriate functionality.

### 3.2. Component Composition

In ArchJava, software architecture is expressed with *composite components*, which are made up of a number of subcomponents<sup>1</sup> connected together. Figure 2 shows how a compiler’s architecture can be expressed in ArchJava. The example shows that the `parser` communicates with the `scanner` using one protocol, and with the

<sup>1</sup> Note: the term *subcomponent* indicates composition, whereas the term *component subclass* would indicate inheritance.

code generator using another. The architecture also implies that the scanner does *not* communicate directly with the code generator. A primary goal of ArchJava is to ease program understanding tasks by supporting this kind of reasoning about program structure.

#### 3.2.1. Subcomponents

A *subcomponent* is a component instance that is declared inside another component class. Components can invoke methods directly on their subcomponents. However, subcomponents cannot communicate with components external to their containing, or *parent*, component. Thus, communication patterns among components are hierarchical.

Subcomponents are declared using a *component field*—a field of component type inside a component class, declared using the `component` keyword. For example, the `compiler` component class defines `scanner`, `parser`, and `code generator` subcomponents. To enable effective static reasoning about subcomponents, component fields are treated as `protected`, `final`, and not `static`. Subcomponents are automatically instantiated when the containing component is created—programmers can use a `new` expression in the field initializer in order to call a non-default constructor.

#### 3.2.2. Connections

The `connect` primitive connects two or more subcomponent ports together, binding each required method to a provided method with the same name and signature. Connections are symmetric, and several connected components may require the same method. Required methods must be connected to exactly one provided method. However, invoking a broadcast method results in calls to each connected provided method with the same name and signature.

Provided methods can be implemented by forwarding invocations to subcomponents or to the required methods of another port. The semantics of method forwarding and broadcast methods are given in the language reference manual on the ArchJava web site [ACN01]. Alternative connection semantics, such as asynchronous communication, can be implemented in ArchJava by writing custom “smart connector” components that take the place of ordinary connections in the architecture.

### 3.3. Communication Integrity

The compiler architecture in Figure 2 shows that while the `parser` communicates with the `scanner` and `code generator`, the `scanner` and `code generator` do not directly communicate with each other. If the diagram in Figure 2 represented an abstract architecture to be implemented in Java code, it might be difficult to verify the correctness of this reasoning in the implementation. For example, if the `scanner` obtained a reference to the `code generator`, it could invoke any of the `code generator`’s methods, violating the intuition communicated by the architecture. In contrast, programmers can have confidence that an ArchJava architecture accurately represents communication between components, because the language semantics enforce communication integrity.

Communication integrity in ArchJava means that components in an architecture can only call each others’ methods along declared connections between ports. Each component in the architecture can use its ports to communicate with the components to which it is connected. However, a component may not directly invoke the methods of components other than its children, because this

communication may not be declared in the architecture—a violation of communication integrity. For example, a component instance  $A$  may not call the methods of another component instance  $B$  unless  $B$  is  $A$ 's subcomponent, or  $A$  and  $B$  are sibling subcomponents of a common component instance that declares a connection or connection pattern between them.

We now precisely define communication integrity in ArchJava. Let the *execution scope* of component instance  $A$  on the run time stack, denoted  $scope(A)$ , be any of  $A$ 's executing methods and any of the object methods they transitively invoke, until another component's method is invoked.

**Definition 1 [Dynamic Execution Scope]:** Let  $m$  be an executing method with stack frame  $m\mathcal{F}$ . If  $m$  is a component method, then  $m\mathcal{F} \in scope(this)$ . Otherwise,  $m\mathcal{F} \in scope(caller(m\mathcal{F}))$ .

Now we can define communication integrity:

**Definition 2 [Communication Integrity in ArchJava]:** Let  $<$  be the subtyping relation over component classes. A program has communication integrity if, for all run time method calls to a method  $m$  of a component instance  $b$  in an executing stack frame  $m\mathcal{F}$ , where  $m\mathcal{F} \in scope(a)$ , either:

1.  $a = b$ , or
2.  $a = parent(b)$ , or
3.  $parent(a) = parent(b) \wedge$  “connect  $f_1.p_1, \dots, f_n.p_n \in class(parent(a))$   
 $\wedge \exists i, j \in 1..n$  s.t.  $parent(a).f_i = a \wedge$   
 $parent(a).f_j = b \wedge$   
 $m \in requiredmethods(p_i) \wedge$   
 $m \in providedmethods(p_j)$ ”

A technical report on the ArchJava web site [ACN01] formalizes the core of ArchJava based on an extension of Featherweight Java [IPW99], and proves type soundness and communication integrity for the core language.

### 3.4. Dynamic Architectures

The constructs described above express architecture as a static hierarchy of interacting component instances, which is sufficient for a large class of systems. However, some system architectures require creating and connecting together a dynamically determined number of components. ArchJava's constructs that support dynamic architectures are discussed in reports on the ArchJava web site.

### 3.5. Limitations of ArchJava

There are currently a number of limitations to the ArchJava approach. Our technique is presently only applicable to programs written in a single language and running on a single JVM, although the concepts may extend to a wider domain. Architectures in ArchJava are more concrete than architectures in ADLs such as Wright, restricting the ways in which a given architecture can be implemented—for example, inter-component connections must be implemented with method calls. Also, in order to focus on ensuring communication integrity, we do not yet support other types of architectural reasoning, such as reasoning about the temporal order of architectural events, or about component multiplicity.

ArchJava's definition of communication integrity supports reasoning about communication through method calls between

components. Program objects can also communicate through data sharing via aliased objects, static fields, and the runtime system. However, existing ways to control communication through shared data often involve significant restrictions on programming style. Future work includes developing ways to reason about these additional communication channels while preserving expressiveness. Meanwhile, our experience (described below) suggests that rigorous reasoning about architectural control flow can aid in program understanding and evolution, even in the presence of shared data structures.

## 4. Evaluation

In order to determine whether the ArchJava language meets its design goals, we undertook a case study to answer the following experimental questions:

- Can ArchJava express the architecture of a real program of significant complexity?
- How difficult is it to reengineer a Java program in order to express its architecture explicitly in ArchJava?
- Does expressing a program's architecture in ArchJava help or hinder software evolution?

An expanded report on the case study is available on the ArchJava web site [ACN01].

### 4.1. Methodology

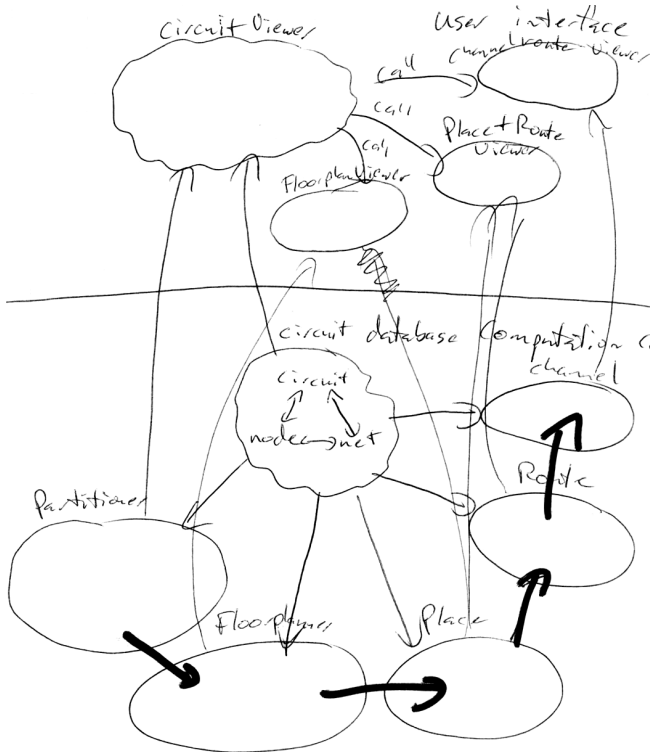
Our approach to answering these questions was to translate a Java program into ArchJava, using the conceptual architecture provided by the program's developer as a guide. In addition to a direct answer to the first two questions for the chosen program and programmer, we hoped to gain some insight into the third question. Other goals included learning about the conceptual architecture of Java programs, gaining practical experience using ArchJava, and refining ArchJava's language design. In the process of our case study, we formed hypotheses for future research, outlined in bold below.

We looked for Java programs that would be at least 10,000 lines of code—large enough that a developer would have difficulty keeping it all in his or her head, and thus might benefit from an explicit software architecture. To avoid biasing our study toward architectures easily expressible in ArchJava, we chose a program and architecture conceived and developed by a third party. Our choice for this case study was the Aphyds program described in the next subsection.

The study's subject (one of us, hereafter “we”) was a graduate student with five year's experience of system programming in Java. Although the subject was the developer of the ArchJava compiler, he was unfamiliar with Aphyds and had little experience writing user interfaces in Java. Thus, the study reflects the common reality of a programmer asked to evolve an unfamiliar system.

We reengineered Aphyds to express the conceptual architecture described by the developer. After browsing the code to determine which classes corresponded to the components in the developer's conceptual architecture, we converted these classes into ArchJava component classes. The resulting architecture was finer grained than the developer's conceptual architecture, so we grouped the component classes into higher-level components.

In order to gain insight into ArchJava's support for software evolution tasks, we performed three experiments. First, we



**Figure 4. The developer’s drawing of Aphyds’ architecture.** The line in the middle of the diagram separates the user interface (above) from the circuit database and computational code (below). Above the line is a set of user interface windows, controlled by the `CircuitViewer`. Below the line are a circuit database of `Node` and `Net` objects and a set of computational modules that act on the circuit database. The unlabeled arrows represent data flow, while the arrows labeled `call` represent the calls relation.

analyzed the inter-component communication patterns in Aphyds, describing and categorizing each different message. Next, we refactored the architecture to simplify and regularize these inter-component communication patterns. Finally, we removed a defect from both the original source code and the ArchJava version of Aphyds.

The next three subsections describe the reengineering process, the software evolution experiments, and how our experience affected the ArchJava language design.

#### 4.2. Reengineering Aphyds

Aphyds, for Academic Physical Design System, is a pedagogical circuit layout application written by an electrical engineering professor for one of his classes. Students are given the program with several key algorithms omitted, and are asked to code the algorithms as assignments. The developer is an experienced programmer with a Ph.D. in computer science, but had no Java background prior to writing Aphyds. The application code is 12,101 lines long, as measured by the Unix `wc` (word count) program, not counting the Java and Semantic libraries used.

Figure 4 shows the developer’s drawing of the conceptual architecture of Aphyds. According to the developer, this abstraction allows him to evolve the system even though the code base is too large to hold in his head at once.

#### 4.2.1. Validating Aphyds’ Architecture

We expected that this architecture would be generally accurate, although it might leave out some details. The developer concurred, saying that all of the links in the architecture are present, but may be subtle to find. Furthermore, the division between UI and functional classes is an important conceptual device for him, but he told us that this division would not necessarily be obvious from looking at the code.

We decided to test this hypothesis by using the Reflexion Model technique [MNS01] to compare the connections in the developer’s conceptual diagram with actual communication patterns between classes in the source code. Overall, the architecture was a good overview of communication in Aphyds. However, the study revealed several minor missing communication paths in the architecture. Moreover, this architecture is also incomplete in some important respects. It does not describe the multiplicity or temporal lifetimes of components. It is at a high level of granularity, as each user interface component represents between 2 and 7 objects. Several complex and messy multi-object communication protocols, dealing with diverse issues, are represented with single lines in the architecture.

Although the developer’s conceptual architecture was informal and flawed in certain respects, this is a realistic example of common practice today. Many developers do not define a formal and precise architecture, but instead communicate the structure of their applications through informal diagrams. One of the motivations for ArchJava is to provide an easy way for developers to gain the benefits of a formal architecture, by embedding it in the code that they write. Our experience with the conceptual architecture of Aphyds is summarized by our first hypothesis, which corroborates findings in the Reflexion Model work [MNS01].

**Hypothesis 1: Developers have a conceptual model of their architecture that is mostly accurate, but this model may be a simplification of reality, and it is often not explicit in the code.**

#### 4.2.2. Reengineering Process

We decided to design a static architecture that follows the developer’s drawing as closely as possible. Therefore, we proposed an Aphyds component to encapsulate the whole application. The Aphyds component would contain the UI components, and would connect them to an AphydsModel component, which would contain a subcomponent for each unit in the lower half of the developer’s diagram. We decided that the `Node` and `Net` objects in the circuit database would remain shared between components; it would have been extremely unnatural to restrict them to within the `Circuit` component.

**Hypothesis 2: Programming languages that prohibit sharing data between components are too inflexible to express the natural architecture for many programs.**

We proceeded to reengineer Aphyds to take advantage of the architectural features of ArchJava. Our technique was to choose one class at a time from the architectural diagram, and turn it into a component class. We started with the `Circuit` class, as this forms the central part of the architectural diagram.

Our task was complicated by the fact that the computational objects in Aphyds were dynamically re-created each time a new circuit was opened. Although ArchJava is capable of expressing this kind of dynamic architecture, we decided to convert the

```

public component class Aphyds {
  // user interface components
  component FloorplanViewer floorplan;
  component ChannelRouteViewer channelRoute;
  component PlaceRouteViewer placeRoute;
  component CircuitViewer viewer;

  // window event communication
  private port window { ... };
  connect window, channelRoute.window,
    viewer.window, placeRoute.window,
    floorplan.window;

  // command protocol
  connect viewer.command, placeRoute.command,
    channelRoute.command, floorplan.command;

  // model components
  component AphydsModel model;

  // protocols for communication with the model
  connect viewer.circuit, placeRoute.circuit,
    model.circuit;
  connect viewer.partition, model.partition;
  connect floorplan.floorplan, model.floorplan;
  connect placeRoute.place, viewer.place,
    model.place;
  connect placeRoute.router, viewer.place,
    model.router;
  connect channelRoute.channel, model.channels;

  // the program's starting point
  public static void main(String args[]) {
    new Aphyds().run();
  }
  public void run() { viewer.setVisible(true); }
}

```

**Figure 5. ArchJava code for the Aphyds components. There are subcomponent declarations for each element in the user interface, as well as a model component that contains the computational code. Connect declarations show communication patterns between components.**

system into a static architecture with components that persisted for the entire execution of the program, believing that this architecture would be simpler to reason about.

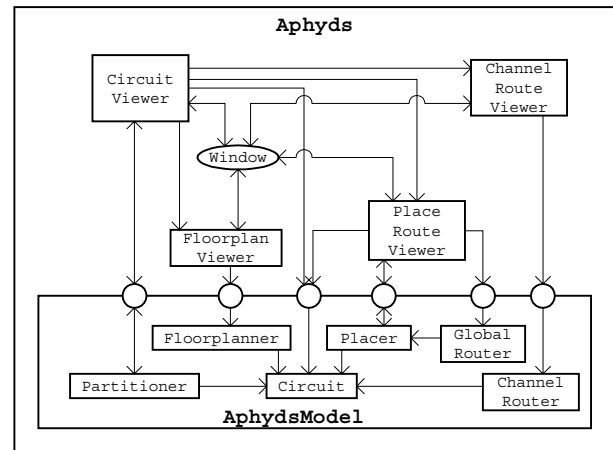
**Hypothesis 3: Describing an existing program’s architecture with ArchJava may involve significant restructuring if the desired architecture does not match the implementation well.**

#### 4.2.3. Reengineering Cost

We spent a total of 30 hours working on Aphyds—15 hours converting the model into components, 8½ hours converting the user interface into components, and 6½ hours refactoring the resulting architecture (as described below). This works out to approximately 2½ hours of work per KLOC. The current code is 12652 lines long—only 551 lines longer than the original application, suggesting that the added architecture code was largely offset by simplifications to the application code.

**Hypothesis 4: Applications can be translated into ArchJava with a reasonable amount of effort, and without excessive code bloat.**

Further study is needed to validate this hypothesis on larger programs, and to determine how the amount of time spent in translation varies with the size of the application and the extent of architectural refactoring required.



**Figure 6. The Aphyds architecture in ArchJava. Squares in the diagram indicate components and arrows represent control flow through connections in the architecture.**

#### 4.2.4. Final Architecture

Figure 5 shows the ArchJava code that expresses the architecture of Aphyds. Compared to the developer’s conceptual architecture, our final ArchJava architecture describes almost identical communication patterns within the circuit database and between the user interface and the database. The multi-way communication between windows that was missing from the original architecture but was present in the program has been consolidated into the window port of Aphyds.

Figure 6 shows a visualization of the current Aphyds architecture manually derived from the ArchJava source code. The developer of Aphyds examined an earlier version of this diagram, and said that it captures his conceptual architecture well, including the separation between the user interface and the circuit database.

The ArchJava architecture has a number of advantages compared to the original, conceptual architecture. ArchJava architectures are guaranteed to be complete, listing all method call communication between components. The ArchJava architecture is guaranteed to stay up-to-date as the code evolves with changing requirements, and a visualization can be generated automatically (although the result may not be as well laid out as Figure 6). Finally, it is easy to zoom in on an ArchJava architecture to look at the interior structure of a component, determine what methods are in each port, or examine how the methods are implemented.

#### 4.2.5. Alternative Architectural Choices

Our study was directed towards implementing the developer’s conceptual architecture as directly as possible in ArchJava. However, an architect could have expressed any of several alternative Aphyds architectures using ArchJava. For example, we could have factored the architecture by functionality, combining each user interface window with the logic that computes the information the window displays. Alternatively, we could have followed the original source code more closely, creating and connecting the model elements on demand as circuits and windows are opened. ArchJava is flexible enough to express these architectures, if the software architect deems them more appropriate.

### 4.3. Software Evolution in ArchJava

In order to gain insight into using ArchJava for software evolution tasks, we examined three concrete problems identified by the developer: understanding communication within the program, refactoring the program to clean up its architecture, and fixing defects related to display updates.

#### 4.3.1. Program Understanding

When we asked the developer if there were any problems with the current structure of Aphyds, he said that communication between the main structures was awkward, especially with respect to change propagation messages. He said that this problem makes it difficult to add new features to the system. This problem had a number of sources: the user interface was partly automatically generated, the developer was new to Java when he started to write the program, and the program grew gradually over time as features were added.

Our experience while reengineering Aphyds corroborated the developer's assertions. Using ad-hoc methods to manually trace method executions was ineffective, because different methods with similar names often did different things, and each method typically depended on the operation of several others. In the original program, the communication patterns were obscure enough that it was hard to analyze and criticize them.

After we initially converted Aphyds to ArchJava, it became clear that the program's communication structure remained inconsistent and unnecessarily complex. Some of these problems had been introduced while refactoring Aphyds to express the architecture, while some were left over from the original source code. However, in the modified program, the port descriptions made communication patterns explicit, and so the communication problems became obvious simply by looking at the methods defined in the ports.

**Hypothesis 5: Expressing software architecture in ArchJava highlights refactoring opportunities by making communication protocols explicit.**

We decided to systematically analyze the communication patterns to find opportunities for refactoring. For each category of messages, we examined the source code to identify the messages' purpose, the message implementers, the message invokers, and the invocation trigger conditions. The communication analysis yielded a number of refactoring opportunities.

ArchJava's language constructs and its guarantee of communication integrity eased our communication analysis. Simply scanning the required and provided methods in each port showed which methods are invoked by and which are implemented by each component. Ports also narrowed our focus to the subset of a component's methods that are involved in inter-component communication. The name of a port also gave a clue about the purpose of the port's methods. Connections showed which other component instances might implement a given component's required methods.

Automated tools could have gathered some of this connectivity information from the original Java program. However, these tools would require sophisticated alias analysis to support the level of reasoning about component instances that is provided by ArchJava's communication integrity. Furthermore, ArchJava makes this connectivity explicit at the source code level, and an

architect can use ports and connections to express design intent in a way that tools cannot duplicate.

**Hypothesis 6: Using separate ports and connections to distinguish different protocols and describing protocols with separate provided and required port interfaces may ease program understanding tasks.**

#### 4.3.2. Fixing Defects

Aphyds' developer said that there were subtle defects in the window update code. To investigate how ArchJava affects the defect-fixing process, we identified and removed a defect that was present both in the original Aphyds code and in the ArchJava version. The defect was observed while changing the location of a circuit element after routing the wires connecting the circuit elements together. The program did not re-compute the routing data, and so the routing display was left in an inconsistent state.

This was a relatively trivial defect, and the solution was the same in both versions. To our surprise, however, fixing the bug in the original Java version was more complex and took longer, because it was difficult to determine how to get a reference to the computational object that performed the update. This task was trivial in the ArchJava version, as the corresponding component was explicit in the architecture.

This defect-fixing example is extremely simple and may not generalize to more complex defects. However, it illustrates the potential of software architecture to ease software evolution tasks by making structure more explicit.

**Hypothesis 7: An explicit software architecture makes it easier to identify and evolve the components involved in a change.**

### 4.4. Effect on the ArchJava Language

While reengineering the Aphyds architecture, we discovered a major shortcoming in the ArchJava language design. To preserve a strong notion of communication integrity, ArchJava's design assumes that control flow originates in components. Components can invoke the methods of objects, but since objects cannot store references to components in their instance fields, they can only invoke methods on components that are passed as arguments to the currently executing method. This creates a significant problem for framework libraries such as the `swing` library used in Aphyds, because these libraries are not written using component classes, yet often they must invoke component methods. This made it impossible to express any meaningful architecture for Aphyds, since all of the application's control flow is driven by the user interface.

Initially, we decided to extend the language by allowing port declarations within objects, and permitting components to make connections between objects and their own subcomponents. This had the crucial advantage of allowing us to work incrementally, transforming one class at a time into a component class by connecting its ports to ports of the surrounding objects. In our reengineering process, we made the database classes into component classes, and initially left the user interface classes as they were, adding ports for communication channels that led to the database. However, the thorniest architectural problems in Aphyds were in the user interface interactions, and since we didn't make the user interface classes into components, our architecture didn't help with these problems at all.

In order for ArchJava to aid our reasoning about communication within the user interface, we decided to also allow component classes to extend regular classes and interfaces, so that legacy libraries could invoke the inherited methods of components through references to the appropriate superclass. These components can then be invoked arbitrarily through their inherited interfaces, threatening communication integrity. However, the new methods introduced in these components can only be called through declared connections in the architecture. These are the methods that express the application logic that we felt was essential to capture and reason about with software architecture. This solution allowed us to convey the architecture of the user interface much more effectively, and was responsible for a disproportionate amount of the software engineering benefits we observed.

#### 4.5. Case Study Summary

We were able to capture the conceptual architecture of Aphyds effectively in ArchJava with a small amount of effort relative to the size of the program. The language made the architecture explicit, and expressing communication protocols through ports helped to clean up communication in the program. The ArchJava compiler helped us in the restructuring task by enforcing communication integrity: it wouldn't let us forget any communication backdoors between components.

### 5. Conclusion and Future Work

ArchJava allows programmers to express architectural structure and then seamlessly fill in the implementation with Java code. At every stage of the software lifecycle, ArchJava enforces communication integrity, ensuring that the implementation conforms to the specified architecture. A case study suggests that ArchJava can be applied to moderate sized Java programs with relatively little effort, resulting in a program structure that more closely matches the designer's conceptual architecture. Thus, ArchJava helps to promote effective architecture-based design, implementation, program understanding, and evolution.

In future work, we intend to gather experience from outside users of ArchJava, and perform further case studies to see if the language can be successfully applied to programs larger than 100,000 lines of code. We will also investigate extending the language design to enable more advanced architectural reasoning, including temporal ordering constraints on component method invocations and constraints on data sharing between components.

### 6. Acknowledgements

We would like to thank Vibha Sazawal, Todd Millstein, Vassily Litvinov, Matthai Philipose, and the anonymous reviewers for their comments. We especially thank Scott Hauck for his time and the Aphyds program. This work was supported in part by NSF grant CCR-9970986, NSF Young Investigator Award CCR-945776, and gifts from Sun Microsystems and IBM.

### 7. References

[ACN01] Jonathan Aldrich, Craig Chambers, and David Notkin. ArchJava web site. <http://www.cs.washington.edu/homes/jonal/archjava/>

[AG97] Robert Allen and David Garlan. A Formal Basis for Architectural Connection. *ACM Transactions on Software Engineering and Methodology*, 6(3):213--249, July 1997.

[B95] Jeremy Buhler. The Fox Project. *ACM Crossroads* 2.1, September 1995.

[FBB+99] Martin Fowler, Kent Beck, John Brant, William Opdyke, and Don Roberts. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999.

[GS93] David Garlan and Mary Shaw. An Introduction to Software Architecture. In *Advances in Software Engineering and Knowledge Engineering, I* (Ambriola V, Tortora G, Eds.) World Scientific Publishing Company, 1993.

[IPW99] Atsushi Igarishi, Benjamin Pierce, and Philip Wadler. Featherweight Java: A minimal core calculus for Java and GJ. In *Proceedings of the Conference on Object Oriented Programming Systems, Languages, and Applications*, November 1999.

[LH89] Karl Lieberherr and Ian Holland. Assuring Good Style for Object-Oriented Programs. *IEEE Software*, Sept 1989.

[LV95] D.C. Luckham, J. Vera. An Event Based Architecture Definition Language. *IEEE Transactions on Software Engineering* Vol. 21, No 9, September 1995.

[MFH01] Sean McDirmid, Matthew Flatt and Wilson C. Hsieh. Jiazi: New-Age Components for Old-Fashioned Java. In *Proceedings of the Conference on Object Oriented Programming Systems, Languages, and Applications*, October 2001.

[MNS01] Gail C. Murphy, David Notkin, and Kevin J. Sullivan. Software Reflexion Models: Bridging the Gap Between Design and Implementation. To appear in *IEEE Transactions on Software Engineering*, 2001.

[MOR+96] Nenad Medvidovic, Peyman Oreizy, Jason E. Robbins, and Richard N. Taylor. Using Object-Oriented Typing to Support Architectural Design in the C2 Style. *Proceedings of SIGSOFT'96: The Fourth Symposium on the Foundations of Software Engineering (FSE-4)*, San Francisco, CA, October 16-18, 1996.

[MQR95] M. Moriconi, X. Qian, A.A. Riemenschneider. Correct Architecture Refinement. *IEEE Transactions on Software Engineering*, Vol. 21, No 4, April 1995.

[MT00] Nenad Medvidovic and Richard N. Taylor. A Classification and Comparison Framework for Software Architecture Description Languages. *IEEE Transactions on Software Engineering*, 26(1):70-93, January 2000.

[MTH90] R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML*. The MIT Press, Cambridge, Mass., 1990.

[PW92] Dewayne E. Perry and Alexander L. Wolf. *Foundations for the Study of Software Architecture*. *ACM SIGSOFT Software Engineering Notes*, 17:40--52, October 1992.

[SC00] J. C. Seco and L. Caires. A Basic Model of Typed Components. *Proc. European Conference on Object-Oriented Programming*, 2000.

[SDK+95] M. Shaw, R. DeLine, V. Klein, T.L. Ross, D.M. Young, G. Zelesnik. Abstractions for Software Architecture and Tools to Support Them. *IEEE Transactions on Software Engineering*, Vol. 21, No 4, April 95.

[Sre01] V. C. Sreedhar. ACOEL: A Component-Oriented Extensional Language. Unpublished manuscript, July 2001.