ArchJava:
Connecting Software Architecture to Implementation

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

Software architecture describes the structure of a system, enabling more effective design, program understanding, and formal analysis. 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 components, connections between the components, and constraints on how the components interact. Describing architecture in a formal architecture description language (ADL) [MT00] can aid in the specification and analysis of high-level designs. Software architecture can also facilitate the implementation and evolution of large software systems. For example, a system's architecture can show which components a module may interact with, help identify the components involved in a change, and describe system invariants that should be respected during software evolution.

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] connect components that are implemented in a separate language. However, these languages do not guarantee that the implementation code obeys architectural constraints. Instead, they require 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. Thus, it may be difficult to trace architectural features to the implementation, and the implementation may become inconsistent with the architecture as the program evolves. In summary, while architectural analysis in existing ADLs may reveal important architectural properties, 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 extension to Java that integrates software architecture specifications smoothly into 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 guarantees communication integrity between an architecture and its implementation, even in the presence of advanced architectural features like run time component creation and connection.

To help 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 that communication. Our experience suggests that the resulting program may be easier to understand and evolve than the original program.

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

2. Previous Work

Architecture Description Languages. A number of architecture description languages (ADLs) have been defined to describe, model, check, and implement software architectures [MT00].

ICSE '02 May 2002 Orlando, FL, USA

Many of these languages support sophisticated analysis and reasoning. For example, Wright [AG97] allows architects to specify temporal communication protocols and check properties such as deadlock freedom. SADL [MQR95] formalizes architectures in terms of theories, shows how generic refinement operations can be proved correct, and describes a number of flexible refinement patterns. Rapide [LV95] supports event-based behavioral specification and simulation of reactive architectures. ArchJava's architectural specifications are probably most similar to those of Darwin [MK96], an ADL designed to support dynamically changing distributed architectures.

While Wright and SADL are pure design languages, other ADLs have supported implementation in a number of ways. UniCon's tools [SDK+95] generate code to connect components implemented in other languages, while C2 [MOR+96] provides runtime libraries in C++ and Java that connect components together. Rapide architectures can be given implementations in an executable sub-language or in languages such as C++ or Ada. More recently, the component-oriented programming languages ComponentJ [SC00] and ACOEL [Sre02] extend a Java-like base language to explicitly support component composition.

However, existing ADLs cannot enforce communication integrity. Instead, system implementers must follow *style guidelines* that ensure communication integrity. For example, the Rapide language manual suggests that components should only communicate with other components through their own interfaces, and interfaces should not include references to mutable types. These guidelines are not enforced automatically and are incompatible with common programming idioms such as shared mutable data structures.

Module Interconnection Languages. Module interconnection languages (MILs) support system composition from separate modules [PN86]. Jiazzi [MFH01] is a component infrastructure for Java, and a similar system, Knit, supports component-based programming in C. These tools are derived from research into advanced module systems, exemplified by MzScheme's Units [FF98] and ML's functors. ADLs differ from MILs in that the former make *connectors* explicit in order to describe *data and control flow* between components, while the latter focus on describing the *uses* relationship between modules [MT00]. Existing MILs cannot be used to describe dynamic architectures, where component object instances are created and linked together at run time.

Furthermore, MILs provide encapsulation by hiding names, which is insufficient to guarantee communication integrity in general. For example, first-class functions or objects can be passed from one module to another, and later used to communicate in ways that are not directly described in the MIL description. Thus, in these systems, programmers must follow a careful methodology to ensure that each module communicates only with the modules to which it is connected in the architecture.

CASE tools. 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 sharing objects between components. This restriction ensures communication integrity, but it also makes the language awkward for general-purpose programming. Many UML tools such as Rational Rose RealTime 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.

Other tools. 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. Similar systems include Virtual Software Classifications [MW99] and Gestalt [SSW96]. Unlike ArchJava, these systems describe architectural components in terms of source code, not run-time component object instances, and the architectural descriptions must be updated separately as the code evolves.

Component Infrastructures. Component-based infrastructures such as COM, CORBA, and Java Beans provide sophisticated services such as naming, transactions and distribution for component-based applications. While these infrastructures do not include mechanisms for explicitly describing software architecture, the Arabica environment [RN00] supports C2 architectures built from off the shelf Java Beans components. This system shows how software architecture can be expressed in the context of component infrastructures, but verifying communication integrity of a Java Beans implementation is left to future work.

3. The ArchJava Language

ArchJava is intended 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.

A prototype compiler for ArchJava is publicly available for download at the ArchJava web site [Arc02]. Although in ArchJava the source code is the canonical representation of the architecture, visual representations are also important for conveying architectural structure. Parts of this paper use 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. Reports on the ArchJava web site [Arc02] provide more

```
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);
  }

  AST 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.

information, including the complete language semantics and a formal proof of communication integrity in the core of ArchJava.

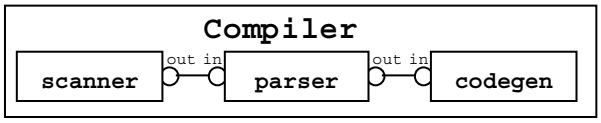
3.1. Components and Ports

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` in Figure 1.

A component can only communicate with other components at its level in the architecture through explicitly declared ports—regular method calls between components are not allowed. A *port* represents a logical communication channel between a component and one or more components that it is connected to.

Ports declare three sets of methods, specified using the **`requires`**, **`provides`**, and **`broadcasts`** keywords. A *provided* method is implemented by the component and is available to be called by other components connected to this port. Conversely, each *required* method is provided by some other component connected to this port. A component can invoke one of its required methods by sending a message to the port that defines the required method. For example, the `parse` method calls `nextToken` on the parser's `in` port. *Broadcast* methods are just like required methods, except that they can be connected to any number of implementations and must return **`void`**.

The goal of this port design is to specify 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.




```
public component class Compiler {
  private final Scanner scanner = ...;
  private final Parser parser = ...;
  private final 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.

ArchJava supports design with **`abstract`** components and ports, which allow an architect to specify and typecheck an ArchJava architecture before beginning program implementation.

3.2. Component Composition

In ArchJava, hierarchical software architecture is expressed with *composite components*, which are made up of a number of subcomponents connected together. A *subcomponent*[1] is a component instance nested within another component. Singleton subcomponents are typically declared with **`final`** fields of component type. 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 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.

Connections. The symmetric **`connect`** primitive connects two or more ports together, binding each required method to a provided method with the same name and signature. The arguments to connect may be a component's own ports, or those of subcomponents in **`final`** fields. Connection consistency checks are performed to ensure that each required method is bound to a unique provided method.

Provided methods can be implemented by forwarding invocations to subcomponents or to the required methods of another port. The

[1] Note: the term *subcomponent* indicates composition, whereas the term *component subclass* would indicate inheritance.

detailed semantics of method forwarding and broadcast methods are given in the language reference manual on the ArchJava web site [Arc02]. 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 other's 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 own subcomponents, because this communication may not be declared in the architecture—a violation of communication integrity. We describe how communication integrity is enforced in section 3.5.

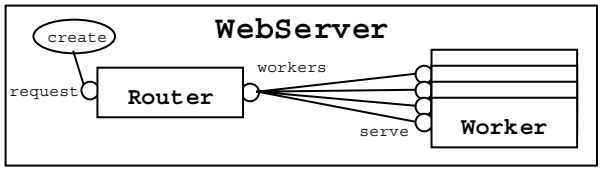
3.4. Dynamic Architectures

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.

Dynamic Component Creation. Components can be dynamically instantiated using the same **`new`** syntax used to create ordinary objects. For example, Figure 2 shows the compiler's `main` method, which creates a `Compiler` component and calls its `compile` method. At creation time, each component records the component instance that created it as its *parent component*. For components like `Compiler` that are instantiated outside the scope of any component instance, the parent component is **`null`**.

Communication integrity places restrictions on the ways in which component instances can be used. Because only a component's parent can invoke its methods directly, it is essential that typed references to subcomponents do not escape the scope of their parent component. This requirement is enforced by prohibiting component types in the ports and public interfaces of components, and prohibiting ordinary classes from declaring arrays or fields of component type. Since a component instance can still be freely passed between components as an expression of type `Object`, a `ComponentCastException` is thrown if an expression is downcast to a component type outside the scope of its parent component instance.

Connect Expressions. Dynamically created components can be connected together at run time using a *connect expression*. For instance, Figure 3 shows a web server architecture where a




```
public component class WebServer {
  private final Router r = new Router();
  connect r.request, create;
  connect pattern Router.workers, Worker.serve;

  private port create {
    provides r.workers requestWorker() {
      final Worker newWorker = new Worker();
      r.workers connection
        = connect(r.workers, newWorker.serve);
      return connection;
    }
  }
  public void run() { r.listen(); }
}

public component class Router {
  public port interface workers {
    requires void httpRequest(InputStream in,
                              OutputStream out);
  }
  public port request {
    requires this.workers requestWorker();
  }
  public void listen() {
    ServerSocket server = new ServerSocket(80);
    while (true) {
      Socket sock = server.accept();
      this.workers conn = request.requestWorker();
      conn.httpRequest(sock.getInputStream(),
                       sock.getOutputStream());
    }
} }

public component class Worker extends Thread {
  public port serve {
    provides void httpRequest(InputStream in,
                              OutputStream out) {
      this.in = in; this.out = out; start();
    }
  }
  public void run() {
    File f = getRequestedFile(in);
    sendHeaders(out);
    copyFile(f, out);
  }
  // more method & data declarations...
}
```

Figure 3. A web server architecture. The `Router` subcomponent accepts HTTP requests and passes them on to a set of `Worker` components that respond. When a request comes in, the `Router` requests a new worker connection on its `request` port. The `WebServer` then creates a new worker and connects it to the `Router`. The `Router` assigns requests to `Workers` through its `workers` port.

`Router` component receives incoming HTTP requests and passes them through connections to `Worker` components that serve the request. The `requestWorker` method of the web server dynamically creates a `Worker` component and then connects its `serve` port to the `workers` port on the `Router`.

Communication integrity requires each component to explicitly document the kinds of architectural interactions that are permitted between its subcomponents. A *connection pattern* is used to describe a set of connections that can be instantiated at run time using connect expressions. For example, **`connect pattern`** `Router.workers, Worker.serve` describes a set of connections between the `Router` subcomponent and dynamically created `Worker` subcomponents.

Each connect expression must match a connection pattern declared in the enclosing component. A connect expression *matches* a connection pattern if the connected ports are identical and each connected component instance is an instance of the type specified in the pattern. The connect expression in the web server example matches the corresponding connection pattern because the `r` and `newWorker` components in the connect expression have static types `Router` and `Worker`, as declared in the pattern.

Port Interfaces. Often a single component participates in several connections using the same conceptual protocol. For example, the `Router` component in the web server communicates with several `Worker` components, each through a different connection. A *port interface* describes a port that can be instantiated several times to communicate through different connections.

Each port interface defines a type that includes all of the required methods in that port. A *port interface type* combines a port's required interface with an *instance expression* that indicates which component instance the port belongs to. For example, in the `Router` component, the type **`this`**`.workers` refers to an instance of the `workers` port of the current `Router` component. Similarly, in the `WebServer`, the type `r.workers` refers to an instance of the `workers` port of the `r` subcomponent. Port interface types can be used in method signatures such as `requestWorker` and in local variable declarations such as `conn` in the `listen` method. In ArchJava, the required methods of a port can only be called by the component instance the port belongs to. Therefore, required methods can only be invoked on expressions of port interface type when the instance expression is **`this`**, as shown by the call to `httpRequest` within `Router.listen`.

Port interfaces are instantiated by connect expressions. A connect expression returns a *connection object* that represents the connection. This connection object implements the port interfaces of all the connected ports. Thus, in Figure 3, the connection object `connection` implements the interfaces `newWorker.serve` and `r.workers`, and can therefore be assigned to a variable of either type.

Provided methods can obtain the connection object through which they were invoked using the **`sender`** keyword. The detailed semantics of **`sender`** and other language features are covered in the ArchJava language reference available on the ArchJava web site [Arc02].

Removing Components and Connections. Just as Java does not provide a way to explicitly delete objects, ArchJava does not provide a way to explicitly remove components and connections. Instead, components are garbage-collected when they are no longer reachable through direct references or connections. For example, in Figure 3, a `Worker` component will be garbage collected when the reference to the original worker (`newWorker`) and the references to its connections (`connection` and `conn`) go out of scope, and the thread within `Worker` finishes execution.

3.5. Enforcing Communication Integrity

An ArchJava architecture shows all of the control-flow paths between sibling components in a subsystem. Thus, communication integrity requires that all inter-component method calls in ArchJava must follow architectural connections, except method calls from a component to its children.

There are two ways in which communication integrity could be violated, corresponding to the two kinds of method calls in ArchJava: direct method calls, and method calls through ports. Below, we describe intuitively how communication integrity is enforced in each of these cases. This intuition corresponds closely to the actual proof of communication integrity in a formal version of ArchJava [Arc02], which we omit for space reasons.

Direct Method Calls. A *direct component method call* is a method call that dispatches to a component instance. The *sending component* of the method call is the receiver of the most recent component method on the stack.

Communication integrity in ArchJava requires that all direct component method calls made from a sending component instance C are to C itself, or to an immediate subcomponent of C. In ArchJava, the only way to directly invoke a component method is to call a method on an expression of component type. ArchJava enforces communication integrity by ensuring the invariant that all of the component-typed expressions in the scope of a component instance C refer to C itself or to an immediate subcomponent of C.

In our formal system, we prove this invariant by induction over program execution, with a case analysis of component type introductions. The invariant is true in the base case of component creation, because a component's parent is by definition the component that created it. Component types can only be read from or written to fields of the current component **`this`**, so they are guaranteed to be children of **`this`** by the induction hypothesis. Similarly, component types cannot be passed directly between components. The final case is casts to component type; but in ArchJava, component casts dynamically check that the cast expression's parent component is the current component **`this`**, ensuring that the invariant continues to hold.

Method Calls through Ports. A connection pattern in a component instance C allows method calls between immediate subcomponents of C through the connected ports. The ArchJava compiler uses a two-part test to verify that all method calls through ports conform to a connection pattern. First, it checks that every connect expression in a component C matches a connect pattern declared in C; the invariant described above ensures that the connected component instances are subcomponents of C. Second, the compiler uses port interface types to verify that each connection object can only be used by the component instances it connects. In ArchJava's type system, only the component instance named in the port interface type is permitted to make calls through the connection object that implements that port interface type. Thus, ArchJava's type system prohibits method calls that would violate architectural constraints.

3.6. Limitations of ArchJava

There are currently several 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 should extend to any statically typed language. 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, because of our focus on ensuring communication integrity, we do not yet support other types of architectural reasoning, such as reasoning about connection protocols, architectural styles, or component multiplicity.

ArchJava's definition of communication integrity supports reasoning about communication through method calls between components. Components can also communicate through shared data or the runtime system. Because existing ways to control communication through shared data involve significant restrictions on programming style, we chose not to adopt these mechanisms. Future work includes developing ways to reason about communication through shared data 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?

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 italics 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 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 Symantec 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.

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. To each of the developer's conceptual components, we assigned one or more implementation classes. We ignored library classes as well as data structures shared by the whole application. We compared the call graph computed by a simple tool to the arrows in the developer's diagram, reversing the direction of his dataflow arrows to reflect control flow in the opposite direction.

Overall, the architecture was a good overview of communication in Aphyds. However, the study revealed several minor missing communication paths in the architecture. For example, although

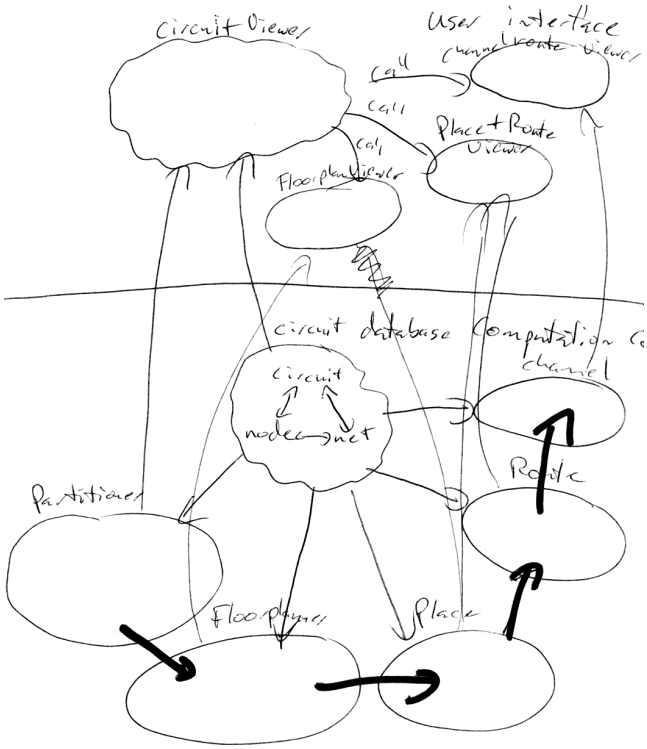


Figure 4. The developer's drawing of Aphyds's architecture. The architecture follows the Model-View design pattern, with the user interface above the line in the middle of the diagram and the circuit database and computational code below. The user interface consists of the `CircuitViewer` window and several subsidiary windows. 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 control flow.

most calls in the application go from the user interface into the model, we found two callbacks going the opposite direction. We also discovered that the communication paths between the `CircuitViewer` and the other viewer objects were actually bi-directional.

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.

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. We expected that this process would primarily consist of converting instance variables into ports or subcomponents, invoking methods on ports instead of instance variables, and connecting the ports appropriately in the architecture.

The structure of the Aphyds implementation made this task more difficult. We initially believed that the architectural drawing represented a set of objects whose membership didn't change over the course of program execution. This was in fact true of the user interface, but the circuit database and computational components were re-created each time they were read from a file or executed. There were a number of methods that set instance variables in the user interface to point to these components; however, many of these methods also had side effects such as refreshing the screen.

We decided to convert the system into a static architecture with components that persisted for the entire execution of the program. Our rationale was that this architecture would be simpler to reason about than a dynamically changing architecture. Therefore, we transformed Aphyds to re-initialize old circuit data structures instead of creating new data structures each time the circuit was loaded. We also separated out the refresh logic from the instance-variable setting messages, so that the architectural connections could be set up at startup time, but the display would still work properly throughout program execution. This reengineering process introduced a number of subtle bugs, partly because we did not recognize the dual nature of these messages until partway through the study.

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

Reengineering Cost. Due to the complexity of separating out the `Circuit` component and our initial unfamiliarity with the application, this first reengineering step took a significant amount of time—about 9½ programmer hours, including time to fix several injected defects.

```
public component class Aphyds {
  // user interface components
  final FloorplanViewer floorplan = ...;
  final ChannelRouteViewer channelRoute = ...;
  final PlaceRouteViewer placeRoute = ...;
  final 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
  final 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);}
}

public component class AphydsModel {
  final Circuit circuitData = ...;
  final Partitioner partitioner = ...;
  final Floorplanner floorplanner = ...;
  final Placer placer = ...;
  final GlobalRouter globalRouter = ...;
  final ChannelRouter channelRouter = ...;

  public port place { ... }
  public port partition { ... }
  public port floorplan { ... }
  public port circuit { ... }
  public port router { ... }
  public port channels { ... }

  connect circuit, partitioner.circuit,
    floorplanner.circuit, placer.circuit,
    globalRouter.circuit, circuitData.main,
    channelRouter.circuit;
  connect place, globalRouter.place,
    placer.place;
  connect partition, partitioner.partition;
  connect floorplan, floorplanner.floorplan;
  connect router, globalRouter.router;
  connect channels, channelRouter.channels;
}
```

Figure 5. ArchJava code for the `Aphyds` and `AphydsModel` 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.

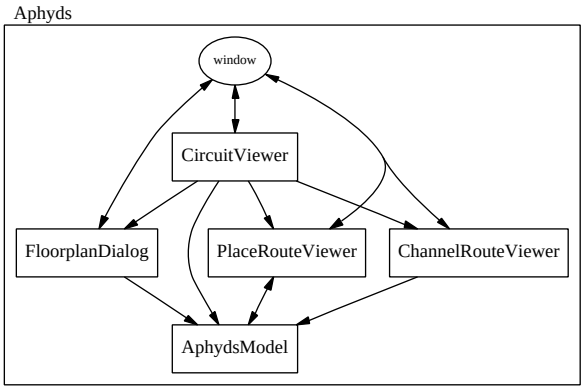


Figure 6. A visualization of the Aphyds architecture, automatically derived from the ArchJava source code. Boxes represent subcomponents, and arrows represent inter-component control flow. The oval denotes the window port, used for window management messages like screen refresh. The circuit database and computational code in the developer's diagram have been isolated in the `AphydsModel` component.

One of the reasons this task may have been difficult is that it was done in a single large step, involving significant application restructuring. An important refactoring principle is to test a program repeatedly while making incremental changes, rather than making a large change all at once [FBB+99]. If we had first transformed the code into an equivalent Java program with a static structure, and only then converted `Circuit` into a component class, we might have been able to detect and repair injected defects earlier and at a smaller cost.

Hypothesis 4: Refactoring an application to expose its architecture is done most efficiently in small increments.

We found support for this hypothesis when transforming the remaining classes into components. These smaller tasks went quickly, taking between 30 and 90 minutes each. 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 5: Applications can be translated into ArchJava with a modest 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.

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, generated automatically from the ArchJava 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. 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.

Alternative Architectural Choices. In our study, we tried to implement 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.

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 6: 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.

ArchJava's language constructs and its guarantee of communication integrity eased this 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 7: Using separate ports and connections to distinguish different protocols and describing protocols with separate provided and required port interfaces may ease program understanding tasks.

Refactoring Architectural Communication. The communication analysis yielded a number of refactoring opportunities. For example, the window refresh logic had been identified by the developer as troublesome in the original application. We found that there were several different refresh methods, each of which affected a subset of the windows. We refactored these into one refresh method that accepted a list of windows to refresh, and modified the method call sites to refresh only the windows affected by the surrounding code.

We found another refactoring opportunity in the data invalidation code. When a new circuit is loaded into the program, data computed about the old circuit must be invalidated. Originally, this was done from many different places in the user interface code, using different message protocols. First, we refactored the invalidation methods to give them consistent names and semantics, and then we simplified the user interface code by moving the invalidation logic from the user interface into the model.

After this refactoring step, communication in Aphyds was considerably easier to understand. Refactoring eliminated a number of methods and even entire categories of communication. The communication categories in the user interface that remained after refactoring include *menu update*, *window refresh*, and *open/close/show window* messages. Between the user interface and the model, our communication categories were *user interface callback*, *command*, *data query*, *data update*, and *validity check* messages.

We could have refactored the program to make these message categories explicit by defining separate ports and connections for each category. In the end, we chose not to do this because the user's conceptual architecture divided up communication according to the computational task, rather than the type of message.

Architectural Refactoring during Translation. While reengineering Aphyds to express the developer's architecture, we found that ArchJava's communication integrity rules forced us to refactor problematic code. For example, class `ChannelRouteDialog` enabled a menu item as follows:

```
getDisplayer().getViewer()
        .ChannelRouterMenuItem.setEnabled(b);
```

This code traverses a series of object links before calling a method on the final object. It violates a design principle known as the Law of Demeter [LH89], which states, "objects should only talk to their immediate neighbors in a system." Code like this makes a program fragile, because this line may break if any object in the sequence of links is changed.

In ArchJava, this code violates communication integrity, because it makes a method call across architectural boundaries. Therefore, during our reengineering, we were forced to refactor this code to call a required method on a local port, which was connected through the architecture to the code that enables the menu item.

Hypothesis 8: Communication integrity in ArchJava encourages local communication and helps to reduce coupling between components.

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 occurred whenever the user changed the location of one element in a routed circuit. 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: we added a call to the `doGlobalRouting` function from the code that moved the circuit element. We repaired the defect in the ArchJava version first. The repair involved adding a `router` port to the component that moves the circuit element, calling `doGlobalRouting` on that port, and connecting the port to the model in the architecture.

Fixing the bug in the original Java version was conceptually simpler, since we didn't have to create or link up the extra port. To our surprise, however, the operation actually turned out to be more complex and took longer, because it was difficult to figure out how to get a reference to the `GlobalRouter` object. The following code shows the complex chain of objects we had to traverse to fix this bug:

```
getDisplayer().placeroutedialog1.placeRouteDisplay
er1.getCircuitGlobalRouter().doGlobalRouting();
```

This defect-fixing example is extremely simple and may not generalize to more complex defects. The comparison above is confounded by many factors, including the order in which the defects were repaired, the confusing user interface source code in the original program, and our familiarity with the two versions of the source code. However, it illustrates the potential of software architecture to ease software evolution tasks by making structure more explicit.

Hypothesis 9: 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. The inherited methods of 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.

To support these new language features, we designed three modes of operation for the ArchJava compiler. A *strict* mode prohibits all of these extensions, and rigidly enforces communication integrity. This mode could be used with future ArchJava user-interface libraries, modeled after the architecture-centric user interface paradigms explored by the C2 project [MOR+96]. An *evolution* mode allows ports to be defined in regular classes, enabling an architecture specification to be added incrementally to a legacy code base such as Aphyds. Finally, a *legacy* mode allows component classes to inherit from ordinary classes and interfaces (with warnings that can be turned off). This mode also permits

inner classes within components, which are important for smooth interoperation with the `swing` library.

Our experience with Aphyds also motivated the design of broadcast methods, and suggested other usability improvements.

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 and suggestions. We also 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

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

[Arc02] Jonathan Aldrich, Craig Chambers, and David Notkin. ArchJava web site. http://www.archjava.org/

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

[FF98] M. Flatt and M. Felleisen. Units: Cool modules for HOT languages. Proc. Programming Language Design and Implementation, Montreal, Canada, June 1998.

[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.

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

[LV95] David C. Luckham and James Vera. An Event Based Architecture Definition Language. IEEE Trans. Software Engineering 21(9), September 1995.

[MFH01] Sean McDirmid, Matthew Flatt and Wilson C. Hsieh. Jiazzi: New-Age Components for Old-Fashioned Java. Proc. Object Oriented Programming Systems, Languages, and Applications, Tampa, FL, October 2001.

[MK96] Jeff Magee and Jeff Kramer. Dynamic Structure in Software Architectures. Proc. Foundations of Software Engineering, San Francisco, CA, October 1996.

[MNS01] Gail C. Murphy, David Notkin, and Kevin J. Sullivan. Software Reflexion Models: Bridging the Gap Between Design and Implementation. IEEE Trans. Software Engineering, 27(4), April 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. Proc. Foundations of Software Engineering, San Francisco, CA, October 1996.

[MQR95] Mark Moriconi, Xiaolei Qian, and Robert A. Riemenschneider. Correct Architecture Refinement. IEEE Trans. Software Engineering, 21(4), April 1995.

[MT00] Nenad Medvidovic and Richard N. Taylor. A Classification and Comparison Framework for Software Architecture Description Languages. IEEE Trans. Software Engineering, 26(1), January 2000.

[MW99] Kim Mens and Roel Wuyts. Declaratively Codifying Software Architectures using Virtual Software Classifications. Proc. Technology of Object-Oriented Languages and Systems Europe, Nancy, France, June 1999.

[PN86] Ruben Prieto-Diaz and James Neighbors. Module Interconnection Languages. Journal of Systems and Software 6(4), April 1986.

[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.

[RN00] David S. Rosenblum and Rema Natarajan. Supporting Architectural Concerns in Component-Interoperability Standards. IEE Proceedings-Software 147(6), 2000.

[SC00] João C. Seco and Luís Caires. A Basic Model of Typed Components. Proc. European Conference on Object-Oriented Programming, Cannes, France 2000.

[SDK+95] Mary Shaw, Rob DeLine, Daniel V. Klein, Theodore L. Ross, David M. Young, and Gregory Zelesnik. Abstractions for Software Architecture and Tools to Support Them. IEEE Trans. Software Engineering, 21(4), April 1995.

[Sre02] Vugranam C. Sreedhar. Mixin' Up Components. Proc. International Conference on Software Engineering, Orlando, FL, May 2002.

[SSW96] Robert W. Schwanke, Veronika A. Strack, and Thomas Werthmann-Auzinger. Industrial software architecture with Gestalt. Proc. International Workshop on Software Specification and Design, Paderborn, Germany, March 1996.