Architectural Reasoning in ArchJava[1]

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 that cause confusion, violate architectural properties, and inhibit software evolution. We are developing ArchJava, an extension to Java that seamlessly unifies software architecture with an object-oriented implementation. In this paper, we show how ArchJava's type system ensures that implementation code conforms to architectural constraints. A case study applying ArchJava to an Islamic tile design application demonstrates that ArchJava can express dynamically changing architectures effectively within implementation code, and suggests that the resulting program may be easier to understand and evolve.

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

[1] This is an extended version of the paper by the same name that appears in the proceedings of the European Conference on Object-Oriented Programming, Málaga, Spain, June 2002. In addition to the complete formalization and proofs for the ArchFJ language, this report makes a few minor corrections and clarifications.

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 conform to its architecture. Luckham and Vera [LV95] identified three criteria for architectural conformance:

- *Decomposition*: For each component in the architecture, there should be a corresponding component in the implementation.
- *Interface Conformance*: Each component in the implementation must conform to its architectural interface.
- *Communication Integrity*: Each component in the implementation may only communication directly with the components to which it is connected in the architecture.

ADLs that provide tool support for skeleton code generation or component linking generally support the first two architectural conformance criteria: decomposition and interface conformance. However, existing ADLs cannot enforce communication integrity, seriously compromising the benefits of architecture during implementation, testing, and software evolution.

We are developing ArchJava [ACN02], a small, backwards-compatible extension to Java that integrates software architecture smoothly with Java implementation code. ArchJava supports a flexible object-oriented programming style, allowing data sharing and supporting dynamic architectures where components are created and connected at run time. The unique feature of ArchJava is a type system that guarantees communication integrity between an architecture and its implementation, even in the presence of shared objects and run-time architecture configuration. In previous work [ACN02] we introduced the ArchJava language and described our initial experience with the subset of ArchJava that supports static architectures.

This paper makes two novel contributions:

- A formalization of the language semantics as ArchFJ, a core language that integrates primitive object-oriented constructs with support for specifying dynamic software architectures. We outline a proof of type soundness and communication integrity for the core language.
- An evaluation of ArchJava in a case study specifying the dynamic architecture of Taprats, a 12,000-line application for designing Islamic tiling patterns.

The rest of this paper is organized as follows. After the next section's discussion of related work, section 3 describes the ArchJava language. Section 4 formalizes ArchJava as ArchFJ, and proves type soundness and communication integrity. Section 5 describes a case study in which we reengineered Taprats, using ArchJava to express

a conceptual architecture drawn by the developer. Finally, section 6 concludes with a discussion of 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]. 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 evolving 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, UML-RT, 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. The techniques described in this paper can be applied in tools such as Rational Rose RealTime to provide a static guarantee of communication integrity.

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.

Previous ArchJava Work. In previous work [ACN02], we describe in detail a case study applying ArchJava to Aphyds, a 12,000-line circuit design application with a static architecture and little use of inheritance. The primary contributions of that paper are an informal description of the language and an empirical evaluation of ArchJava on the Aphyds application. In contrast, this paper contributes a formalization of the language design, and proofs of type soundness and communication integrity. This paper also presents a new case study applying ArchJava to Taprats, a second, contrasting application that exercises ArchJava's support for dynamic architectures and component inheritance.

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 object-oriented 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

```
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);
  }

  public void parse() {
    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 enters tokens into 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.

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 3.5.

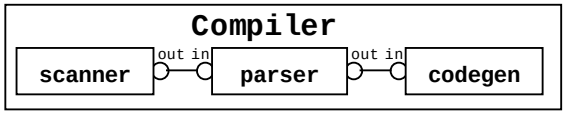
To allow programmers to describe software architecture, ArchJava adds new language constructs to support *components*, *connections*, and *ports*. The rest of this section reviews the language design [ACN02], describing by example how to use these constructs to express software architectures. Throughout the discussion, we show how the constructs work together to enforce communication integrity. Reports on the ArchJava web site [Arc02] provide more information, including the complete language semantics.

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.

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




```
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();...
  }
}
```

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.

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 an unbounded 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.

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*[2] is a component instance nested within another component. Singleton subcomponents can be declared as **`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 detailed semantics of method forwarding and broadcast methods are given in the language reference manual on the ArchJava web site [Arc02].

ArchJava does not explicitly support alternative connection semantics such as asynchronous communication; however, these semantics can be implemented in ArchJava by writing custom components that play the role of "smart connectors." The ArchJava release includes an example `AsynchronousConnector` component that caches required method calls in an internal worklist and then returns immediately, invoking the corresponding provided methods asynchronously from an internal thread.

Inheritance. Component classes can inherit from other component classes, or from class `Object`. The compiler's legacy mode also allows component classes to inherit from ordinary classes, at the cost of losing communication integrity guarantees for inherited methods, so that developers can use non-component-based legacy frameworks like the Java GUI libraries. Component subclasses inherit methods, ports, and connections from their superclasses. Component subclasses may also override method definitions and specify new methods and ports. However, component subclasses may not specify new required methods because this could break subtype substitutability.

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

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

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

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.

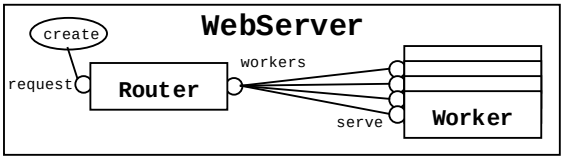
Communication integrity in ArchJava means that components in an architecture can only call each other's methods along declared connections between ports. Each component in the architecture can use its ports to communicate with the components to which it is connected. However, a component may not directly invoke the methods of components other than its own subcomponents, because this communication may not be declared in the architecture, and thus may violate communication integrity. We define communication integrity more precisely in section 4.1.

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 *container component*. For components like `Compiler` that are instantiated outside the scope of any component instance, the container component is **`null`**.

Communication integrity places restrictions on the ways in which component instances can be used. Because only a component's container can invoke its methods directly, it is essential that typed references to subcomponents do not escape the scope of their container 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 container component instance.




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

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

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() {
    // gets requested file and sends it on the output stream
  }
}
```

Figure 3. A web server architecture. The `Router` subcomponent accepts incoming 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.

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 `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 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 conform to the types `Router` and `Worker` that are declared in the connection 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 use the **`sender`** keyword to obtain the connection object through which they were invoked. 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, running threads, or architectural 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. 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 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, our design focuses on ensuring communication integrity, and does 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; however, components may still used shared data to communicate in ways that are not directly expressed in the architecture. Because existing ways to control communication through shared data involve significant restrictions on programming style, we chose to allow unrestricted data sharing. Future work includes developing ways to reason about communication through shared data while preserving expressiveness. Our preliminary experience with ArchJava [ACN02] suggests that rigorous reasoning about architectural control flow can aid in program understanding and evolution, even in the presence of shared data structures.

3.6. Implementation

A prototype compiler for ArchJava is publicly available for download at the ArchJava web site [Arc02]. Our compiler is implemented on top of the Barat infrastructure [BS98]. The compiler accepts a list of ArchJava files (.archj), compiles each one down to Java source code, and invokes `javac` on the resulting .java files. Our compilation technique is incremental, so that when a source file is updated, only that file and the files that depend on it need to be recompiled.

The ArchJava compiler translates each component class to an ordinary class in Java, leaving the fields and method bodies substantially unchanged. We mangle the

names of classes to ensure that code not compiled by our compiler will not accidentally misuse component classes; the `main` function is left in a class with the original name, so that ArchJava applications work smoothly on existing Java virtual machines. Each component class stores its container component and implements an interface that allows the container to be checked. All casts to a component class are compiled into calls to a generated cast method that verifies that the cast expression's container is the current component **`this`**, throwing a `ComponentCastException` if the check fails.

Each port and port interface in the ArchJava source code is compiled into an interface containing the required methods of the port. All variables of port interface type are compiled to variables of that port's interface type.

Each connection is compiled into a "connection class" that implements all of the interfaces of the connected ports. The connect expression returns a new connection object, passing the connected components to the connection object's constructor. The constructor assigns the connected components to internal fields. Whenever a required method is invoked on that connection, the connection object invokes the corresponding provided method on the appropriate component.

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.

Performance. The main cost of our implementation technique is that calls through connections are routed through connection objects, adding a layer of indirection to the system. Our current compiler is a prototype and does not perform any optimizations; however, future implementations could use well-known techniques like specialization to eliminate this indirection in many cases.

Thus far, the only applications of significant size to which we have applied ArchJava are interactive, and thus it is difficult to benchmark their performance. An independent evaluation of ArchJava on a microbenchmark that exhibited a very fine-grained architecture measured an overhead of about 10% relative to Java code with a similar decomposition [AL02]. We expect that most realistic applications would use architectural features at a more coarse grain, and so this estimate is probably close to the worst case in practice.

4. ArchJava Formalization

In this section, we discuss the formal definition of communication integrity and ArchJava's semantics. The next subsection defines communication integrity in ArchJava. Subsection 4.2 gives the static and dynamic semantics of ArchFJ, a language incorporating the core features of ArchJava. Finally, subsection 4.3 outlines

states type soundness theorems for ArchFJ, and subsection 4.4 outlines a proof of communication integrity.

4.1. Definition of Communication Integrity

Communication integrity is the key property of ArchJava that ensures that the implementation does not communicate in ways that could violate reasoning about control flow in the architecture. Intuitively, communication integrity in ArchJava means that 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. We want to reason about not only the direct method calls made by a component instance c to a component instance b, but also indirect method calls made through non-component intermediary objects. To reason about these indirect calls, we define the *execution scope* of component instance c on the run-time stack, denoted *escope*(c), be any of c's executing methods and any of the object methods they transitively invoke.

Definition 1 [Dynamic Execution Scope]: Let c be a component instance, let mf range over method frames executing on the stack, and assume that the *caller* function returns the previous method frame on the stack. Then we can define the execution scope of c recursively as follows:

$$\begin{aligned} \mathit{escope}(\mathtt{c}) \quad &\equiv \quad \{\ \mathtt{mf} \mid (\mathtt{mf.this} = \mathtt{c})\ \} \cup \\ &\qquad \{\ \mathtt{mf} \mid !\mathit{component}(\mathtt{mf.this}) \wedge \mathit{caller}(\mathtt{mf}) \in \mathit{escope}(\mathtt{c})\ \} \\ \mathit{escope}(\mathtt{null}) \quad &\equiv \quad \{\ \mathtt{mf} \mid \forall \mathtt{c} \neq \mathtt{null}\ .\ \mathtt{mf} \notin \mathit{escope}(\mathtt{c})\ \} \end{aligned}$$

It is easy to show that each method frame mf is in the execution scope of either exactly one component or **null**.

Now we can define communication integrity. Let <: be the subtyping relation over component classes (defined precisely in section 4.2, below). Let the function *container* return a component's container component (i.e., the component instance in whose scope it was created), or **null** if there is no such container. We use *class* to refer to the class of a component instance, and *requiredmethods* and *providedmethods* to refer to the set of required and provided methods in a port.

Definition 2 [Communication Integrity in ArchJava]: 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 mf ∈ *escope*(a):

1. For direct method calls, a = b or a = *container*(b)
2. For calls through connections, there exists a component instance c such that:
   - c = a or c = *container*(a), and
   - c = b or c = *container*(b), and
   - **connect pattern** $P_1.z_1, \ldots, P_n.z_n \in$ *class*(c), and
   - $\exists i,j \in 1..n$. *class*(a) <: $P_i \wedge$ *class*(b) <: $P_j \wedge$ m ∈ *requiredmethods*(p_i) ∧ m ∈ *providedmethods*(p_j)

In the definition above, the first case represents direct method calls between components: the callee must either be the caller itself or one of the caller's subcomponents. The second case represents a method call along a connection between components: some component instance c that is equal to or contains a and b must have declared a connection pattern between a and b that matches the types of a and b and includes the invoked method m. In section 4.4, we state and prove this communication integrity property for a core subset of ArchJava.

This definition has been simplified slightly in the interest of clarity. Calls to broadcast methods can be modeled as calls to multiple required methods, and static connections can be modeled with dynamic connections and connect pattern declarations.

4.2. Formalization as ArchFJ

We would like to use formal techniques to prove that the ArchJava language design guarantees communication integrity, and show that the language is type safe. A standard technique, exemplified by Featherweight Java [IPW99], is to formalize a core language that captures the key typing issues while ignoring complicating language details. We have formalized ArchJava as ArchFJ, a core language based on Featherweight Java (FJ).

Syntax. Figure 4 presents the syntax of ArchFJ. The metavariables C and D range over class names; E and F range over component and class names; T and V range over types; P and Q range over component classes; f and g range over fields; v ranges over values; d and e range over expressions; z ranges over port interface names; S ranges over stores; ℓ, θ, and ς range over locations in the store, where θ is typically used to represent the value of **this** and ς represents the container of an object, and M ranges over methods. As a shorthand, we use an overbar to represent a sequence. We assume a fixed class table *CT* mapping regular and component classes to their definitions. A program, then, is a pair (*CT*, e) of a class table and an expression.

ArchFJ makes a number of simplifications relative to ArchJava. In ArchFJ, each component has exactly one port interface, defining a set of required and provided methods. For simplicity, we require that the same set of methods appear in the class

```
CL   ::= class C extends C {C f; K M}
CP   ::= component class P extends E {C f; K M port interface z {R M} X}
K    ::= E(C f) {super(f); this.f = f;}
M    ::= T m(T x) { return e; }
R    ::= requires T m(T x)
X    ::= connect pattern (P.z)

e    ::= v
     |   new E(e)
     |   e.f
     |   e.m(e)
     |   (T)e
     |   θ ▷ e

v    ::= x
     |   ℓ
     |   connect(v.z)
     |   error

T,V  ::= E
     |   v.z
     |   U(v.z)

S    ::= ℓ → E_ς(ℓ)
Γ    ::= x → T
Σ    ::= ℓ → T

ℓ, θ, ς ∈ Locations
```

Figure 4. ArchFJ Syntax

body and in the port body. Static connections and ports are left out, as they are subsumed by dynamically created connections and port interfaces. We also omit the **sender** keyword and broadcast methods. As in Featherweight Java (FJ), we omit interfaces, assignment, and some statement and expression forms. These changes make our type soundness proof shorter, but do not materially affect it otherwise.

ArchFJ extends FJ in several ways. Regular classes extend another class (which can be `Object`, a predefined class) and define a constructor `K` and a set of fields $\overline{f}$ and methods $\overline{M}$. Component classes can extend another component class, or `Object`. Component classes also define a single port interface that includes a set of required methods $\overline{R}$ and provided methods $\overline{M}$. Finally, component classes declare a set of connection patterns $\overline{X}$ between their subcomponents.

We need to reason about object identity (represented by a location ℓ) in order to verify communication integrity. A store `S` maps locations ℓ to their contents: the class of the object and the values stored in its fields. As in ArchJava, the store also keeps track of each object's container object (represented by a subscript on the class name) in order to check run time component casts properly. We will write `S[`ℓ`]` to denote

$$\frac{\ell \notin \mathit{domain}(S) \qquad S' = S[\ell \to E_\theta(\bar{\ell})]}{S, \theta \vdash \textbf{new}\ E(\bar{\ell}) \to \ell,\ S'} \qquad \text{(R-NEW)}$$

$$\frac{S[\ell] = E_\varsigma(\bar{\ell}) \qquad \mathit{fields}(E) = \bar{C}\ \bar{f}}{S, \theta \vdash \ell.f_i \to \ell_i,\ S} \qquad \text{(R-FIELD)}$$

$$\frac{S[\ell] = F_\varsigma(\bar{\ell}) \qquad F <: E \qquad E\ \mathit{a\ component} \Rightarrow (\theta = \ell \vee \theta = \varsigma)}{S, \theta \vdash (E)\ell \to \ell,\ S} \qquad \text{(R-CAST)}$$

$$\frac{\ell.z \in \bar{\ell}.\bar{z}}{S, \theta \vdash (\ell.z)(\textbf{connect}(\bar{\ell}.\bar{z})) \to \textbf{connect}(\bar{\ell}.\bar{z}),\ S} \qquad \text{(R-CONNECTCAST)}$$

$$\frac{S[\ell] = E_\varsigma(\bar{\ell}) \qquad \mathit{mbody}(m, E) = (\bar{x}, e_\theta) \qquad e_b = [\bar{v}/\bar{x},\ \ell/\textbf{this}]e_\theta}{S, \theta \vdash \ell.m(\bar{v}) \to \ell \triangleright e_b,\ S} \qquad \text{(R-INVK)}$$

$$\frac{S[\bar{\ell}] = \bar{P}_{\bar{\varsigma}}(\ldots) \qquad \mathit{mbody}(m, \bar{P}) = (\bar{x}, e_\theta, i) \qquad e_b = [\bar{v}/\bar{x},\ \ell_i/\textbf{this}]e_\theta}{S, \theta \vdash \textbf{connect}(\bar{\ell}.\bar{z}).m(\bar{v}) \to \ell_i \triangleright e_b,\ S} \qquad \text{(R-CONNECTINVK)}$$

$$S, \theta \vdash \ell \triangleright v \to v,\ S \qquad \text{(R-CONTEXT)}$$

Figure 5. ArchFJ Evaluation Rules

the store entry for ℓ. Functional store updates are abbreviated $S[\ell \to E_\ell(\bar{\ell})]$. The function *container*(S,ℓ) looks up the container ς of ℓ in store S.

Expressions include object creation expressions, field lookup, method calls, and casts. Component creations and casts must refer to the current value of **this**, so our reduction rules keep track of the **this** reference as part of the executing context. We give a small-step reduction semantics, and so a program expression must represent a stack of executing methods, each with a potentially different receiver value **this**. Therefore, we use an expression $\theta \triangleright e$ to represent a method body e executing with a receiver θ.

Values represent irreducible computational results, and include locations and connections. ArchFJ represents failed casts with an explicit **error** value. We also include variables as values because a variable may appear as the instance expression in a port interface type. The set of variables includes the distinguished variable **this** used to refer to the receiver of a method. Neither the **error** value, nor locations, nor $\theta \triangleright e$ expressions may appear in the source text of the program; these forms are only generated during reduction.

Types include class and component types (E), port interface types (v.z), and a union type that matches any one of a set of port interface types.

Reduction Rules. The evaluation relation, defined by the reduction rules given in Figure 5, is of the form $S, \theta \vdash e \to e', S'$ read "In the context of store S and receiver θ, expression e reduces to expression e′ in one step, producing the new store S′." We write $\to^*$ for the reflexive, transitive closure of $\to$. Most of the rules are

$$\frac{S,\theta \vdash e_i \rightarrow e_i', S'}{S,\theta \vdash \textbf{new}\ E(v_1 \ldots v_{i-1}, e_i \ldots e_n) \rightarrow \textbf{new}\ E(v_1 \ldots v_{i-1}, e_i', e_{i+1} \ldots e_n), S'} \quad \text{(RC-NEW)}$$

$$\frac{S,\theta \vdash e \rightarrow e', S'}{S,\theta \vdash e.f_i \rightarrow e'.f_i, S'} \quad \text{(RC-FIELD)}$$

$$\frac{S,\theta \vdash e \rightarrow e', S'}{S,\theta \vdash (E)e \rightarrow (E)e', S} \quad \text{(RC-CAST)}$$

$$\frac{S,\theta \vdash e \rightarrow e', S'}{S,\theta \vdash (v.z)e \rightarrow (v.z)e', S'} \quad \text{(RC-CONNECTCAST)}$$

$$\frac{S,\theta \vdash e \rightarrow e', S'}{S,\theta \vdash e.m(\bar{e}) \rightarrow e'.m(\bar{e}), S'} \quad \text{(RC-RECVINVK)}$$

$$\frac{S,\theta \vdash e_i \rightarrow e_i', S'}{S,\theta \vdash v.m(v_1 \ldots v_{i-1}, e_i \ldots e_n) \rightarrow v.m(v_1 \ldots v_{i-1}, e_i', e_{i+1} \ldots e_n), S'} \quad \text{(RC-ARGINVK)}$$

$$\frac{S,\ell \vdash e \rightarrow e', S'}{S,\theta \vdash \ell \triangleright e \rightarrow \ell \triangleright e', S'} \quad \text{(RC-CONTEXT)}$$

$$\frac{S[\ell] = F_\varsigma(\bar{\ell}) \qquad F \not<: E \vee (E\ \textit{a component} \wedge \theta \neq \ell \wedge \theta \neq \varsigma)}{S,\theta \vdash (E)\ell \rightarrow \textbf{error}, S} \quad \text{(E-CAST)}$$

$$\frac{\ell.z \notin \bar{\ell}.\bar{z}}{S,\theta \vdash (\ell.z)(\textbf{connect}(\bar{\ell}.\bar{z})) \rightarrow \textbf{error}, S} \quad \text{(E-CONNECTCAST)}$$

Figure 6. ArchFJ Congruence and Error Rules

standard; the interesting features are how they manipulate architectural constructs. The R-NEW rule reduces a new expression to a fresh location. The store is updated at that location to refer to a new object with its fields set to the values passed into the constructor, and with its container set to the current object θ (representing **`this`**). The field read rule looks up the receiver in the store and returns the location in the *i*th field.

The reduction rule for object casts looks up the actual type of the casted object in the store, and verifies that the actual type is a subtype of the type in the cast expression. In addition, if the cast is to a component class, the rule verifies that the current component θ is equal to either ℓ or ℓ's container ς. Similarly, the rule for casts to a port interface type verifies that the named port interface type is one of the ones in the actual connection. If any of the conditions on the cast fails, then the cast reduces to the **`error`** expression (these *cast error rules* are shown in Figure 6).

The method invocation rule R-INVK looks up the receiver in the store, then uses the *mbody* helper function (defined in Figure 10) to determine the correct method body to invoke. In the method body, all occurrences of the formal method parameters and **`this`** are replaced with the actual arguments and the receiver, respectively. Execution of the method body continues in the context of the receiver location. The

$$\mathtt{T} <: \mathtt{T} \qquad \text{(S-REFLEX)}$$

$$\frac{\mathtt{T} <: \mathtt{T}' \qquad \mathtt{T}' <: \mathtt{T}''}{\mathtt{T}' <: \mathtt{T}''} \qquad \text{(S-TRANS)}$$

$$\frac{CT(\mathtt{E}) = [\mathtt{component}]\ \mathtt{class\ E\ extends\ F\ \{\ldots\}}}{\mathtt{E} <: \mathtt{F}} \qquad \text{(S-EXTENDS)}$$

$$\mathtt{T} <: \mathtt{Object} \qquad \text{(S-OBJECT)}$$

$$\frac{\mathtt{v.z} \in \overline{\mathtt{v.z}}}{\bigcup(\overline{\mathtt{v.z}}) <: \mathtt{v.z}} \qquad \text{(S-UNION)}$$

Figure 7. ArchFJ Subtyping

rule for invocations on connections is similar, except that the *mbody* helper function also determines which of the connected components defines the invoked method. When a method expression reduces to a value, the R-CONTEXT rule propagates the value outside of its method context and into the surrounding method expression.

Figure 6 shows the congruence rules that allow reduction to proceed within an expression in the order of evaluation defined by Java. For example, the rule RC-FIELD states that an expression `e.f` reduces to `e′.f` whenever `e` reduces to `e′`. The congruence rule RC-CONTEXT shows the semantics of the $\ell \triangleright \mathtt{e}$ construct: evaluation of the expression `e` occurs in the context of the receiver ℓ instead of the receiver θ. The error rules define the semantics of a failed cast. Whenever the run-time checks necessary for a cast fail, we reduce the cast expression to an **`error`** value, which is how we model the exception that is thrown by the full language when a cast fails.

Subtyping Rules. ArchFJ's subtyping rules are given in Figure 7. Subtyping of classes and components is defined by the reflexive, transitive closure of the immediate subclass relation given by the **`extends`** clauses in *CT*. In the S-EXTENDS rule and elsewhere, the brackets and ellipses indicate optional syntax that does not affect the rule's semantics. We require that there are no cycles in the induced subtype relation. Every type is a subtype of `Object`, and a union type is a subtype of all its member types.

Typing Rules. Typing judgments, shown in Figure 8, are of the form $\Gamma,\Sigma,\mathtt{E} \vdash \mathtt{e}:\mathtt{T}$, read "In the type environment Γ, store typing Σ, and receiver class E, expression `e` has type `T`." The T-VAR rule looks up the type of a variable in Γ, and the T-LOC rule looks up the type of a location in Σ. The object creation rule verifies that the types of all the actual constructor argument types are subtypes of the declared constructor argument types (which match the class's fields).

The connection rule assigns the connection a union type of all the connected ports. If the instance expressions in the connection are variables, then this is a connection in the source text, and so the connection must match a connect pattern declaration in the enclosing component. It is not necessary to perform this check once the variables in

$$\Gamma,\Sigma,E \vdash \ell : \Sigma(\ell) \qquad \text{(T-Loc)}$$

$$\Gamma,\Sigma,E \vdash x : \Gamma(x) \qquad \text{(T-Var)}$$

$$\frac{\Gamma,\Sigma,F \vdash \overline{e} : \overline{C} \qquad \mathit{fields}(E) = \overline{D}\ \overline{f} \qquad \overline{C} <: \overline{D}}{\Gamma,\Sigma,F \vdash \textbf{new}\ E(\overline{e}) : E} \qquad \text{(T-New)}$$

$$\frac{\begin{array}{c}\Gamma,\Sigma,E \vdash \overline{v} : \overline{P} \\ \overline{v} = \overline{x} \Rightarrow (E = P_{this} \wedge \textbf{connect pattern}\ \overline{Q}.\overline{z} \in \mathit{connects}(P_{this}) \wedge \overline{P} <: \overline{Q})\end{array}}{\Gamma,\Sigma,E \vdash \textbf{connect}(\overline{v}.\overline{z}) : \cup(\overline{v}.\overline{z})} \qquad \text{(T-Connect)}$$

$$\frac{\Gamma,\Sigma,E \vdash e_0 : E_0 \qquad \mathit{fields}(E_0) = \overline{C}\ \overline{f}}{\Gamma,\Sigma,E \vdash e_0.f_i : C_i} \qquad \text{(T-Field)}$$

$$\frac{\Gamma,\Sigma,E \vdash e : T_0 \qquad T \textit{ is a component} \Rightarrow E \textit{ is a component}}{\Gamma,\Sigma,E \vdash (T)e : T} \qquad \text{(T-Cast)}$$

$$\frac{\begin{array}{c}\Gamma,\Sigma,E_{this} \vdash e_0 : T_0 \qquad \mathit{mtype}(m,T_0) = \overline{T} \to T \qquad \Gamma,\Sigma,E_{this} \vdash \overline{e} : \overline{V} \\ \overline{V} <: [e_0/\textbf{this}]\overline{T} \qquad T_R = [e_0/\textbf{this}]T \qquad T_0 = x.z \Rightarrow x = \textbf{this}\end{array}}{\Gamma,\Sigma,E_{this} \vdash e_0.m(\overline{e}) : T_R} \qquad \text{(T-Invk)}$$

$$\frac{\Gamma,\Sigma,E \vdash \ell : F \qquad \Gamma,\Sigma,F \vdash e : T}{\Gamma,\Sigma,E \vdash \ell \triangleright e : T} \qquad \text{(T-Context)}$$

Figure 8. ArchFJ Typechecking

the connect expression have been replaced with locations, as the property is already ensured. The rule for field reads looks up the declared type of the field using the *fields* function defined in Figure 10. Casts to a component class in ArchJava can only appear in methods of a component class; the cast rule for ArchFJ checks this constraint.

Rule T-INVK looks up the invoked method's type using the *mtype* function defined in Figure 10, and verifies that the actual argument types are subtypes of the method's argument types. Because **`this`** may appear as part of port interface types in the method's argument and result types, the rule substitutes any occurrences of **`this`** in the method's type with the actual receiver value. This substitution is undefined if the method's type contains **`this`** and the receiver is not a value. If the invocation is through a port interface type and the instance expression is a variable, then the instance expression must be **`this`**, as in ArchJava. Finally, the T-CONTEXT typing rule for an executing method checks the method's body in the context of the class of the **`this`** pointer.

Class and Store Typing. Figure 9 shows the rules for well-formed class definitions and stores. The rules for well-formed classes have the form "class declaration E is OK," and "method/port/connection X is OK in E." The class rule checks that the constructor first calls the superclass constructor, and then assigns the values passed to the constructor to the corresponding fields. It also verifies that any methods, ports, and connections in the class are well-formed, and checks our simplifying assumption

$$\frac{\texttt{K} = \texttt{F}(\overline{\texttt{D}}\ \overline{\texttt{g}},\ \overline{\texttt{C}}\ \overline{\texttt{f}})\ \{\textbf{super}(\overline{\texttt{g}});\ \textbf{this}.\overline{\texttt{f}} = \overline{\texttt{f}};\} \qquad \mathit{fields}(\texttt{E}) = \overline{\texttt{D}}\ \overline{\texttt{g}} \qquad \overline{\texttt{M}}\ \texttt{OK IN F} \quad \texttt{F}\ \mathit{a\ component\ class} \Rightarrow ((\texttt{E}\ \mathit{a\ component\ class} \vee \texttt{E} = \texttt{Object}) \wedge \texttt{z}, \overline{\texttt{X}}\ \texttt{OK IN F})}{[\textbf{component}]\ \textbf{class}\ \texttt{F}\ \textbf{extends}\ \texttt{E}\ \{\overline{\texttt{C}}\ \overline{\texttt{f}};\ \texttt{K}\ \overline{\texttt{M}}\ [\textbf{port}\ \texttt{z}\{\overline{\texttt{R}}\ \overline{\texttt{M}}\}\ \overline{\texttt{X}}]\}\ \texttt{OK}} \quad \text{(T-CLASS)}$$

$$\frac{CT(\texttt{E}) = [\textbf{component}]\ \textbf{class}\ \texttt{E}\ \textbf{extends}\ \texttt{F}\ \{\ldots\} \qquad \mathit{override}(\texttt{m},\ \texttt{F},\ \overline{\texttt{T}} \to \texttt{T}) \qquad \{\overline{\texttt{x}}{:}\overline{\texttt{T}},\ \textbf{this}{:}\texttt{E}\}, \varnothing, \texttt{E} \vdash \texttt{e}:\texttt{V} \qquad \texttt{V} <: \texttt{T} \qquad \texttt{T}, \overline{\texttt{T}}\ \mathit{not\ components}}{\texttt{T}\ \texttt{m}(\overline{\texttt{T}}\ \overline{\texttt{x}})\ \{\quad \textbf{return}\ \texttt{e};\ \}\ \texttt{OK in E}} \quad \text{(T-METH)}$$

$$\frac{\texttt{R}_i = \textbf{requires}\ \texttt{T}_i\ \texttt{m}(\overline{\texttt{T}_i}\ \overline{\texttt{x}_i});\qquad \texttt{T}_i, \overline{\texttt{T}_i}\ \mathit{not\ components} \qquad CT(\texttt{P}) = \textbf{component class}\ \texttt{P}\ \textbf{extends}\ \texttt{E}\ \{..\} \qquad \texttt{E} \neq \texttt{Object} \Rightarrow \textbf{port}\ \texttt{z}\ \{\ \overline{\texttt{R}}\ \overline{\texttt{M}}\ \} \in \texttt{E}}{\textbf{port}\ \texttt{z}\ \{\ \overline{\texttt{R}}\ \overline{\texttt{M}}\ \}\ \texttt{OK in P}} \quad \text{(T-PORT)}$$

$$\frac{\forall \texttt{m}, i\ \ (\mathit{mtype}(\texttt{m}, \texttt{z}_i) = \overline{\texttt{T}} \to \texttt{T}) \quad \Rightarrow\ (\exists j \neq i\ \ s.t.\ \ \mathit{mtype}(\texttt{m}, \texttt{P}_j) = \overline{\texttt{T}} \to \texttt{T}\ \ \wedge\ \ \forall k \neq j\ \ \mathit{mtype}(\texttt{m}, \texttt{P}_k)\ \ not\ \ defined)}{\textbf{connect pattern}(\overline{\texttt{P}.\texttt{z}})\ \texttt{OK IN}\ \ \texttt{Q}} \quad \text{(T-PATTERN)}$$

$$\frac{dom(\Sigma) = dom(\texttt{S}) \qquad \forall \ell \in dom(\texttt{S})\ .\ (\texttt{S}[\ell] = \texttt{E}, (\overline{\ell}) \wedge \mathit{fields}(\texttt{E}) = \overline{\texttt{C}}\ \overline{\texttt{f}} \Rightarrow \Sigma(\overline{\ell}) <: \overline{\texttt{C}})}{\Sigma \vdash \texttt{S}} \quad \text{(T-STORE)}$$

Figure 9. Class, Method, Port, Connection, and Store Typing

that a component class's port defines the same set of methods as the class. Component classes may only inherit from other component classes, or from class `Object`.

The rule for methods checks that the method body is well typed, and uses the *override* function (defined in Figure 10) to verify that methods are overridden with a method of the same type. It also ensures that the signature of a component method does not include component types. For component classes, the port typing rule verifies that only subclasses of `Object` may define new required and provided methods. The rule for connect patterns verifies that for each required method there is a unique provided method with the right signature.

The store typing rules ensure that the form of the store is consistent with the Java's typing rules. The two clauses of the store typing rule are the usual well-formedness rules, requiring the store type Σ to type every location in $\texttt{S}$, and verifying that the types of objects in a field are compatible with the field's type.

Auxiliary Definitions. Most of the auxiliary definitions shown in Figure 10 are straightforward and are derived from FJ. The field and connection lookup rules return the list of fields and connections in a given class. ArchFJ follows Java's lookup rules for method types and method bodies, with straightforward extensions for port types and union types. We assume that all port interfaces unrelated by inheritance have distinct names, simplifying our *mtype* rule. The method body lookup rule *mbody* for connections chooses the component *i* providing the method. It is guaranteed to choose a unique component because the T-PATTERN rule implies that only one of the

Field lookup:

$$\frac{}{\mathit{fields}(\texttt{Object}) = \bullet}$$

$$\frac{CT(\texttt{E}) = [\textbf{component}]\ \textbf{class}\ \texttt{E}\ \textbf{extends}\ \texttt{F}\ \{\overline{\texttt{C}}\ \overline{\texttt{f}};\ \texttt{K}\ \overline{\texttt{M}}\ [\overline{\texttt{R}}\ \overline{\texttt{X}}]\} \qquad \mathit{fields}(\texttt{F}) = \overline{\texttt{D}}\ \overline{\texttt{g}}}{\mathit{fields}(\texttt{E}) = \overline{\texttt{D}}\ \overline{\texttt{g}},\ \overline{\texttt{C}}\ \overline{\texttt{f}}}$$

Connection lookup:

$$\frac{}{\mathit{connects}(\texttt{Object}) = \bullet}$$

$$\frac{CT(\texttt{P}) = \textbf{component}\ \textbf{class}\ \texttt{P}\ \textbf{extends}\ \texttt{E}\ \{\overline{\texttt{C}}\ \overline{\texttt{f}};\ \texttt{K}\ \overline{\texttt{M}}\ \overline{\texttt{R}}\ \overline{\texttt{X}}\} \qquad \mathit{connects}(\texttt{E}) = \overline{\overline{\texttt{X}_0}}}{\mathit{connects}(\texttt{P}) = \overline{\overline{\texttt{X}_0}},\ \overline{\texttt{X}}}$$

Method type lookup:

$$\frac{CT(\texttt{E}) = [\textbf{component}]\ \textbf{class}\ \texttt{E}\ \textbf{extends}\ \texttt{F}\ \{\ldots\overline{\texttt{M}}\ldots\} \qquad \texttt{T m}\ (\overline{\texttt{T}}\ \overline{\texttt{x}})\ \{\ \textbf{return}\ \texttt{e};\ \} \in \overline{\texttt{M}}}{\mathit{mtype}(\texttt{m}, \texttt{E}) = \overline{\texttt{T}} \to \texttt{T}}$$

$$\frac{CT(\texttt{E}) = [\textbf{component}]\ \textbf{class}\ \texttt{E}\ \textbf{extends}\ \texttt{F}\ \{\ldots\overline{\texttt{M}}\ldots\} \qquad \texttt{m}\ \textit{is not defined in}\ \overline{\texttt{M}}}{\mathit{mtype}(\texttt{m}, \texttt{E}) = \mathit{mtype}(\texttt{m}, \texttt{F})}$$

$$\frac{\textbf{component}\ \textbf{class}\ \texttt{P}\ \textbf{extends}\ \texttt{E}\ \{\ldots\textbf{port}\ \texttt{z}\ \{\overline{\texttt{R}}\ \overline{\texttt{M}}\}\ldots\} \in CT \qquad \textbf{requires}\ \texttt{T m}\ (\overline{\texttt{T}}\ \overline{\texttt{x}}) \in \overline{\texttt{R}}}{\mathit{mtype}(\texttt{m}, \texttt{z}) = \overline{\texttt{T}} \to \texttt{T}}$$

$$\frac{\mathit{mtype}(\texttt{m}, \texttt{z}_i) = \overline{\texttt{T}} \to \texttt{T}}{\mathit{mtype}(\texttt{m}, \bigcup(\overline{\texttt{v}}.\overline{\texttt{z}})) = \overline{\texttt{T}} \to \texttt{T}}$$

Method body lookup:

$$\frac{CT(\texttt{E}) = [\textbf{component}]\ \textbf{class}\ \texttt{E}\ \textbf{extends}\ \texttt{F}\ \{\ldots\overline{\texttt{M}}\ldots\} \qquad \texttt{T m}\ (\overline{\texttt{T}}\ \overline{\texttt{x}})\ \{\ \textbf{return}\ \texttt{e};\ \} \in \overline{\texttt{M}}}{\mathit{mbody}(\texttt{m}, \texttt{E}) = (\overline{\texttt{x}}, \texttt{e})}$$

$$\frac{CT(\texttt{E}) = [\textbf{component}]\ \textbf{class}\ \texttt{E}\ \textbf{extends}\ \texttt{F}\ \{\ldots\overline{\texttt{M}}\ldots\} \qquad \texttt{m}\ \textit{is not defined in}\ \overline{\texttt{M}}}{\mathit{mbody}(\texttt{m}, \texttt{E}) = \mathit{mbody}(\texttt{m}, \texttt{F})}$$

$$\frac{\mathit{mbody}(\texttt{m}, \texttt{P}_i) = (\overline{\texttt{x}}, \texttt{e}_0)}{\mathit{mbody}(\texttt{m}, \overline{\texttt{P}}) = (\overline{\texttt{x}}, \texttt{e}_0, i)}$$

Valid method overriding:

$$\frac{\mathit{mtype}(\texttt{m},\ \texttt{E}) = \overline{\texttt{T}} \to \texttt{T}_0 \ \Rightarrow\ \overline{\texttt{V}} = \overline{\texttt{T}}\ \wedge\ \texttt{V}_0 = \texttt{T}_0}{\mathit{override}(\texttt{m},\ \texttt{E},\ \overline{\texttt{V}} \to \texttt{V}_0)}$$

Figure 10. ArchFJ Auxiliary Definitions

components in a connection defines each method. It then computes the actual method body using the usual *mbody* rule. Finally, the *override* rule checks that overriding methods have the same type signatures as the methods they override.

4.3. Type Soundness

Before proving communication integrity, we show that our type system is sound, i.e., execution of ArchFJ programs will not become stuck except due to failed casts. We frame type soundness with the usual theorems: Subject Reduction states that if a well-typed program reduces to another program in a single reduction step, the resulting program is either well-typed or contains an **`error`** subexpression from a failed cast. Progress states that a well-typed program is either an irreducible value or an expression to which one or more of the evaluation rules applies. Our presentation is modeled after that of Featherweight Java [IPW99].

Theorem [Subject Reduction]: If $\Gamma,\Sigma,\mathsf{E} \vdash \mathsf{e}:\mathsf{T}$, $\Sigma(\theta)=\mathsf{E}$, $\Sigma \vdash \mathsf{S}$, and $\mathsf{S},\theta \vdash \mathsf{e} \rightarrow \mathsf{e}',\mathsf{S}'$ then either $\exists \Sigma' \supseteq \Sigma$, $\mathsf{T}' <: \mathsf{T}$ such that $\Gamma,\Sigma',\mathsf{E} \vdash \mathsf{e}':\mathsf{T}'$ and $\Sigma' \vdash \mathsf{S}'$, or else e' has an **`error`** subexpression.

Before proving the theorem, we define a term substitution lemma, necessary for the method invocation case in the proof. This enables us to show that substituting terms in a well-typed expression preserves the typing:

Lemma [Term Substitution]: If $\{\overline{\mathsf{x}}:\overline{\mathsf{T}}, \mathbf{this}:\mathsf{E}\}, \emptyset, \mathsf{E} \vdash \mathsf{e}:\mathsf{T}$, $\emptyset,\Sigma,\mathsf{F} \vdash \overline{\ell}:\overline{\mathsf{V}}$, $\emptyset,\Sigma,\mathsf{F} \vdash \ell:\mathsf{E}'$, $\overline{\mathsf{V}} <: [\ell/\mathbf{this}]\overline{\mathsf{T}}$, and $\mathsf{E}' <: \mathsf{E}$, then $\emptyset,\Sigma,\mathsf{F} \vdash \ell \triangleright [\overline{\ell}/\overline{\mathsf{x}},\ell/\mathbf{this}]\mathsf{e}:\mathsf{T}'$ for some $\mathsf{T}' <: [\ell/\mathbf{this}]\mathsf{T}$.

The proof is by induction over the structure of e, with a case analysis on the form of the outermost term. Most of the cases in the proof are trivial; the more interesting cases are as follows:

Case T-LOC: Not applicable since e is given a type in the empty store typing.

Case T-VAR: The variable x must be one of **`this`** or $\overline{\mathsf{x}}$. But these variables were replaced with locations that are given a subtype of the variable types, so the case holds.

Other cases follow trivially from the induction hypothesis and the transitivity of subtyping. □

We then prove subject reduction by induction on the derivation of $\mathsf{S},\theta \vdash \mathsf{e} \rightarrow \mathsf{e}',\mathsf{S}'$ with a case analysis on the outermost reduction rule used (one regular or error reduction rule may apply, in addition to any number of congruence rules).

Case R-NEW: We extend the store type to give ℓ the type E, preserving the type of the expression. We know that the store will remain well-typed because the rule T-NEW ensures that the arguments to the constructor have appropriate types.

Case R-FIELD: Follows from store typing and the rule T-FIELD.

Case R-CAST and R-CONNECTCAST: These reductions only apply if the resulting expression is a subtype of the cast type, so the cases hold.

Case R-INVK: By simultaneous induction over the operation of *mtype* and *mbody*, we can see that the actual method has the type attributed by *mtype*. By applying the rule T-INVK, the term-substitution lemma, and the rule for well-typed methods, the

method body has a subtype of the declared return type (modulo the right substitution for **`this`**). The case is completed by observing that the context construct allows us to type the method body in the context of a new class for **`this`**.

Case R-CONNECTINVK: Similar to R-INVK, but we extend the induction on *mtype* and *mbody* to cover invocations on connect expressions.

Case R-CONTEXT: We rely on the invariant that the context form can only exist when variables have been substituted with locations within the context expression. In this case, type of the value on the right hand side cannot depend on the class of **`this`**, so the case holds.

Subject reduction for the cast error rules follows since these rules reduce to the **`error`** expression, and it follows trivially for the congruence rules by the induction hypothesis. □

Theorem [Progress]: If $\varnothing,\Sigma,\mathsf{E} \vdash \mathsf{e}:\mathsf{T}$, then either e is an irreducible value, or else $\forall \mathsf{S}, \theta$ such that $\Sigma \vdash \mathsf{S}$ and $\Sigma(\theta)=\mathsf{E}$ we have $\mathsf{S},\theta \vdash \mathsf{e} \rightarrow \mathsf{e}',\mathsf{S}'$.

The proof is by induction on the derivation of $\varnothing,\Sigma,\mathsf{E} \vdash \mathsf{e}:\mathsf{T}$ with a case analysis on the last typing rule used.

Cases T-LOC, T-VAR, and T-CONNECT: e is an irreducible value.

Case T-NEW: If the constructor arguments are values, reduction R-NEW applies, otherwise RC-NEW applies by the induction hypothesis. In future cases, we will assume subexpressions are values, as the corresponding congruence rule applies by the induction hypothesis in all cases.

Case T-FIELD: Reduction R-FIELD applies because the induction hypothesis (together with the definition of *fields*) implies that the location ℓ refers to an object that has the field being read.

Case T-CAST: At least one of the cast or cast error reduction rules must apply, as these reduction rules cover all possible cases.

Case T-INVK: The induction hypothesis implies that *mtype* returns a method type based on the type of the receiver location or connect expression. By simultaneous induction on the execution of *mtype* and *mbody*, we can see that *mbody* returns a method body, and so either R-INVK or R-CONNECTINVK applies.

Case T-CONTEXT: Rule R-CONTEXT applies. □

4.4. Communication Integrity

Like the definition of communication integrity for ArchJava in section 4.1, communication integrity for ArchFJ has two parts: a theorem for direct method calls, and a theorem for method calls through a connection. The first theorem states that for all direct method invocations on a component, the receiver must be the current component **`this`** or one of its immediate subcomponents:

Theorem [Communication Integrity of Direct Calls]: For all applications $\mathsf{S},\theta \vdash \ell.\mathsf{m}(\overline{\mathsf{v}}) \rightarrow \ell \triangleright \mathsf{e}_b, \mathsf{S}$ of reduction rule R-INVK in a program execution

$\emptyset, \theta_{init} \vdash e_{init} \rightarrow^* v_{fin}, S_{fin}$ where $\emptyset,\emptyset,\texttt{Object} \vdash e_{init} : T_{init}$ and $S[\ell]=P\ldots$, the current component θ is either the receiver ℓ of the method call or the container of the receiver ($\theta=\ell$ or $\theta=container(S,\ell)$).

Proof: By the type soundness theorem, we know that every step in the derivation of $\emptyset, \theta_{init} \vdash e_{init} \rightarrow^* v_{fin}, S_{fin}$ is well typed. Consider the expression `e.m(…)` that is a subexpression of e_{init} or some method substituted body e_b' such that $S, \theta \vdash e \rightarrow^* \ell, S'$ is a subsequence of $\emptyset, \theta_{init} \vdash e_{init} \rightarrow^* v_{fin}, S_{fin}$. We prove the theorem by a case analysis on the structure of `e`:

Case ℓ: First, note that `e` can only be a location if it arises by substitution in a method call. The substituted variable cannot be of type `Object`, since `Object` has no defined methods in ArchFJ. Since we know that the execution typechecks and that ℓ has a component type, and since ordinary arguments may not have component type, the substituted variable must be **`this`** and so $\theta=\ell$.

Case **`new`** `P(…)`: The last reduction in $S, \theta \vdash e \rightarrow^* \ell, S'$ must be R-NEW, and so $\theta=container(S,\ell)$.

Case `(P')e''`: The last reduction in $S, \theta \vdash e \rightarrow^* \ell, S'$ must be R-CAST, and so $\theta=container(S,\ell)$ or $\theta=\ell$.

Case `e''.f`: Impossible since `e` is given a component type and fields can't hold components.

Case `e''.m(…)`: Impossible since `e` is given a component type and methods can't return components.

Case **`connect`**: Impossible since we are applying rule R-INVK.

Case **`error`**: Impossible since `e` is given a type.

Case `x`: Impossible since `e` already has its variables substituted.

Case $\ell' \triangleright e''$: Impossible since `e` is part of the source text. □

Theorem [Communication Integrity of Indirect Calls]: For all expressions **`connect`**$(\bar{\ell}.\bar{z})$ that arise in a program execution $\emptyset, \theta_{init} \vdash e_{init} \rightarrow^* v_{fin}, S_{fin}$ where $\emptyset,\emptyset,\texttt{Object} \vdash e_{init} : T_{init}$, then there exists a component instance ℓ that declared a connection pattern ($\textbf{connect pattern } \bar{Q}.\bar{z} \in connects(\Sigma(\ell))$) whose types match the connected components ($\Sigma(\bar{\ell}) <: \bar{Q}$), and all of the connected components are equal to or contained by ℓ ($\forall \ell_i \in \bar{\ell} \;.\; \ell = \ell_i \vee \ell = container(S, \ell_i)$). Furthermore, if a method is called on the connection during program execution ($S, \theta \vdash \textbf{connect}(\bar{\ell}.\bar{z}).m(\bar{v}) \rightarrow \ell_i \triangleright e_b, S$) then the current component $\theta \in \bar{\ell}$.

The proof of the first part is by induction on the derivation of $\emptyset, \theta_{init} \vdash e_{init} \rightarrow^* v_{fin}, S_{fin}$. The only reduction steps that affect the theorem are the method invocation rules, which introduce new connect expressions into an executing program. Here, the T-CONNECT rule verifies that the appropriate connection pattern is present in the enclosing component class. A lemma analogous to the communication integrity of direct calls theorem is used to show that $\forall \ell_i \in \bar{\ell} \;.\; \ell = \ell_i \vee \ell = container(S, \ell_i)$.

Finally, we show that $\theta \in \bar{\ell}$. The key insight is that rule T-INVK requires that any port interface type with a variable as the instance expression must be of the form **`this`**`.z`. When the method is called the variable **`this`** will be replaced with the

actual receiver θ, and so type soundness guarantees that the connect expression includes θ. □

5. Experience

In previous work, we validated the basic design of ArchJava with a case study on Aphyds, a 12,000-line circuit-design program with a static architecture [ACN02]. In this section, we describe a case study that evaluates ArchJava's support for dynamic architectures and component inheritance, and adds to our confidence in the application of ArchJava. In our case study, we attempt to answer the following experimental questions:

- Is ArchJava expressive enough to describe a real architecture that is dynamically evolving?
- How does the difficulty of reengineering a Java program in order to express its architecture vary with the program's characteristics?
- What might be the benefits of expressing a program's architecture in ArchJava?

5.1. Methodology

Our approach to answering these questions was to translate Taprats from Java into ArchJava, using the conceptual architecture provided by the program's developer as a guide. In the process of our Taprats case study, we refined the hypotheses formed in our initial case study, and made new hypotheses, outlined in bold below.

The case study participant was a graduate student with five years' experience of system programming in Java. Although the participant was the developer of the ArchJava compiler, he was unfamiliar with Taprats 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. The participant was one of us, and will be informally referred to as "we" in the following text.

We reengineered Taprats 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 next four subsections describe the process of reengineering Taprats, a comparison to the earlier Aphyds case study, an analysis of what we learned about the ArchJava language, and a summary of the benefits of reengineering Taprats in ArchJava.

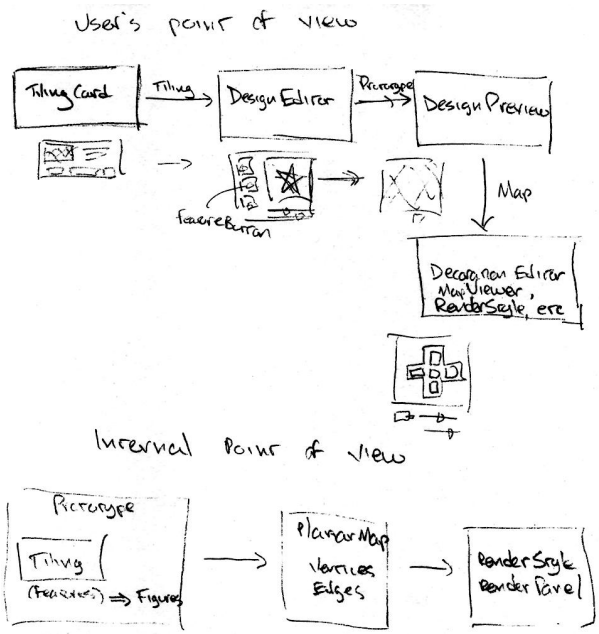


Figure 11. The developer's drawing of Taprats' architecture. The drawing on the top shows the user's point of view, describing the four main user interface windows, what they look like on the screen, and what data structures are passed from one window to the next. The drawing on the bottom shows the internal data structures, beginning with a `Tiling` that is nested withing a `Prototype`, which first evolves into a `Map` and then has rendering style information added.

5.2. Reengineering Taprats

Taprats [Kap00] is an application for designing Islamic star patterns. The user first chooses a basic tiling pattern from a library, then defines the exact shapes used within the tiles, and finally renders the design in one of several styles. Different windows are provided for these tasks, and the user can simultaneously work on different variations of a single design.

The developer of Taprats (not one of us) is a computer science graduate student and an experienced Java programmer. Taprats won the grand prize in the 2000 ACM/IBM Quest for Java, and can thus be considered a model Java program with a quality design and implementation. The application is 12,540 lines of Java source code, as measured by the Unix `wc` (word count) program, not counting the Java libraries used.

We asked the developer to draw the conceptual architecture of Taprats, as shown in Figure 11. He drew two diagrams, one representing the user interface and one representing the internal data structures. The user interface is a pipeline architecture

of four windows, each of which passes an increasingly detailed data structure to the next window. The internal view shows how data structures are contained within and produced from each other.

Validating Taprats' Architecture. We began the study by examining the Taprats source code to try to determine how it corresponds to the developer's conceptual architecture. We discovered that the `main` method in the `Program` class created the first user interface window, and that each successive window spawned the next one in the action code for the appropriate button.

Although the conceptual architecture of the user interface showed a sequence of windows, the implementation structure was more like a nesting of window instances, where each window object is responsible for creating child window objects for the next tile design stage. Thus, our experience with Taprats supports a hypothesis from our previous case study:

> **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.**

Architectural Design Principles. ArchJava provides two kinds of objects with which to build applications. Component objects allow developers to specify the communication patterns within an architecture, but the compiler's communication integrity checks limit the ways in which component objects can be used. ArchJava also provides ordinary Java objects, which allow unrestricted data sharing within a component architecture, but which cannot be used to specify or check architectural properties. Design principles are needed to help determine where to use component objects and where to use ordinary objects.

Using the intuition that architecture is most important at the largest scales in the application, we began our study by creating a component representing the entire Taprats application, and then refined this architecture to increase its level of detail. We used the following guidelines to help us choose which application objects should be components in the architecture, and which are best left as ordinary objects:

- *Scale.* The larger the scale of the component, the more program understanding and evolution benefits may be gained by making its internal structure explicit. This is primarily because other tools for program understanding (including browsing source code) are the least effective at large scales.
- *Control flow.* Does one of the constituent objects of a component call back into that component? If so, that object will have to be made part of the architecture to satisfy the compiler's communication integrity checks. This rule is largely a consequence of ArchJava's focus on control flow communication integrity.
- *Sharing.* ArchJava supports a hierarchical view of software architecture, and therefore does not allow a component to be shared by two container components. Thus, structures that are shared between components should be left as ordinary objects, unless the sharing can be easily replaced with method calls through the container component's port.

- *Database objects.* Singleton objects that encapsulate information shared by multiple components are good component candidates, forming a repository architecture style. They may need to be promoted up a level in the component hierarchy to make the sharing explicit.
- *Data structures.* Small data structures that have many instances and are shared or passed between components are best left as ordinary objects. ArchJava's component mechanisms may be too "heavyweight" to use at these small application scales.
- *Cooperation.* If a set of objects communicate with each other in complex ways, making them component classes in an architecture may aid program understanding by making the communication patterns explicit as connections in the architecture.
- *Lack of communication.* ArchJava's architectural features can be used to document the invariant that a set of components do not communicate directly with one another.

These principles are not orthogonal; a designer must make tradeoffs based on the applicability of the different design criteria, and the specific nature of the application. We hope to refine these design principles based on future experience with ArchJava.

Architectural Design. Applying the design principles above, we initially focused on the architecture of the user interface, as shown in the top part of Figure 10. Our rationale was that the user interface is the highest level of *scale* in the application, and also that *control flow* originates in the user interface. Our experience suggests:

> **Hypothesis 2: Because ArchJava ensures control-flow communication integrity, it has a natural bias towards a UI-centric architecture in user-interface driven applications.**

This hypothesis is also supported by our previous case study, which also resulted in a UI-centric architecture. Our hypothesis suggests that in the future, we should apply ArchJava to systems applications that are not user-interface driven, to determine the effectiveness of the language in that domain.

As we reengineered Taprats, we used the architecture design guidelines to flesh out our initial architecture. Following the developer's conceptual architecture, we made each user interface window into a component. We then refined the architecture by making several window panes into subcomponents of their containing window, either because there was *control flow* from the pane back into the window, or because we wanted to document the fact that the panes were *unshared* and they *did not communicate* with other components. Ultimately, we decided not to encode the bottom part of Figure 11 in the architecture, because these are *data structures* that are passed along the user interface pipeline.

Parts of the user interface architecture made extensive use of inheritance, exercising ArchJava's support for component inheritance. For example, the user interface employs window panes of different classes depending on the tiling pattern chosen by the user. Taprats' design shows how inheritance can be useful in a component-based system.

Code Restructuring. As described above, each window in the user interface creates the next one, suggesting a series of nested windows rather than a pipeline of windows. In order to make the developer's conceptually linear architecture more explicit, we decided to make two structural changes to the application.

First, we made the windows siblings in the architecture instead of being nested within each other. Because components can only be created by their container component in ArchJava, this meant we had to move all the application's window-creation code into the `Program` class. This change complicated the application slightly, because each window had to call into the container component to create the next window. However, it has benefits as well: the new design shows the conceptual architecture more directly than the original design. This "factory pattern" design [GHJ+94] also decouples the different user interface windows, because each window no longer specifies exactly which window will be created next and how it will be created. This information is hidden within the container component, potentially allowing the interface to be modified at a smaller cost.

> **Hypothesis 3: Using ArchJava to express software architecture explicitly can aid information hiding by encouraging developers to reduce coupling between different components in their architecture.**

In a post-study interview, the Taprats developer said that this change made the ArchJava architecture appear more like his conceptual architecture, but thought that there should be some way to allow components to be constructed by their siblings in the architecture. We are considering how to address this limitation of the current ArchJava language design, perhaps by supporting constructor calls through ports.

Second, instead of passing tiling data from one window to the next via an argument to the latter window's constructor, we created explicit connections between the windows, along which the data could be passed. We made this change in order to express the developer's conceptual architecture as directly as possible, and the developer agreed that the new design helped to accomplish this goal. However, a serious drawback of the new design is that windows are not completely initialized when the constructor completes, but remain in a partially initialized state until the tiling data is passed via a separate method call. Because of this, the developer said that he would not have made this second architectural change. It is possible that allowing constructor calls through ports will enable us to express this type of connection directly without the drawbacks of our current implementation.

Reengineering Process. We performed our reengineering as a series of small refactoring steps, compiling the program and fixing introduced defects after every stage. Thus, we never went more than an hour without a correctly running program. This methodology was suggested in our previous case study, after we tried to make many changes at once and ended up introducing several hard-to-repair defects. We found that this methodology was effective at limiting defects in this study.

To help us understand the process of reengineering a program to make its architecture explicit with ArchJava, we recorded the major refactoring steps we performed, and categorized them into the following refactoring patterns:

- *Change class to component class*: When a class describes an object that is part of the architecture, change it into a component class. This may require applying other refactorings in order to pass communication integrity checks.
- *Move creation to container component*: When a component creates one of its sibling components in the architecture, create a port in the component and its container with a single method, `requestCreate`. The container component creates the sibling in `requestCreate`, connects it as appropriate in the architecture, and optionally returns a connected port to the original child component.
- *Change a field link into a connection*: When a component has a field that refers to a sibling component, replace the field with a port that contains all of the methods invoked on the sibling component. In the container component, connect the component's port to a corresponding port on its sibling, and then convert method invocations on the field into invocations on the appropriate port.

In addition to these major refactoring steps, we used several conventional refactoring patterns [FBB+99], as well as a few more minor refactoring patterns that are specific to ArchJava.

Reengineering Cost. We spent about 5½ developer hours reengineering Taprats, or about 30 minutes of work per KLOC. Of this time, approximately half was spent in design activity—understanding the structure of the original program, planning the conversion to ArchJava, considering architectural alternatives, and examining the final architecture for completeness at the end. Because the developer of Taprats had already put considerable effort into making a clean design and implementation, a relatively small amount of our time was spent actually implementing the architectural changes.

Our implementation time was divided roughly equally between modifying the source code to express the architecture, and repairing defects that were introduced in these refactoring steps. The final program code is 12693 lines long—only 153 lines longer than the original application. A total of 242 lines of code were added or changed in the process. Our experience supports a hypothesis from our previous study:

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

Code Characteristics. One particular code characteristic that stood out as we edited Taprats was the Law of Demeter [LH89], which states that objects should only communicate directly with their immediate neighbors in a system. The Law of Demeter can be thought of as the object-oriented analog of communication integrity, since ArchJava components may only communicate with the architectural "neighbors" to which they are connected in the architecture.

```
public component class Program {

  // the tiling selector window subcomponent
  private final TilingSelector ts = new TilingSelector();

  // ports for creating windows
  private port createDesignEditor {
    provides ts.sendTiling requestEditor() {
      DesignEditor e = new DesignEditor();
      connect(createPreviewPanel, e.createNext);
      ts.sendTiling aPort = connect(ts.send, e.receive);
      return aPort;
    }
  }
  private port interface createPreviewPanel {
    provides Object requestPreview(Object edit) {
      DesignEditor e = (DesignEditor) edit;
      PreviewPanel p = new PreviewPanel();
      connect(createRenderPanel, p.createNext);
      return connect(e.send, p.receive);
    }
  }
  private port interface createRenderPanel {
    provides Object requestRender(Object prevw) {
      PreviewPanel p = (PreviewPanel) prevw;
      RenderPanel r = new RenderPanel();
      return connect(p.send, r.receive);
    }
  }

  // connections between the creation ports and the windows
  connect createDesignEditor, ts.createNext;
  connect pattern createPreviewPanel, DesignEditor.createNext;
  connect pattern createRenderPanel, PreviewPanel.createNext;

  // connections between the windows
  connect pattern TilingSelector.send, DesignEditor.receive;
  connect pattern DesignEditor.send, PreviewPanel.receive;
  connect pattern PreviewPanel.send, RenderPanel.receive;

  // the main methods of the program
  public void run() {
    Frame f = new Frame( "Taprats 0.3" );
    f.add( "Center", ts );
    // more code to finish setting up the window...
  }

  public static void main(String[] args) {
    new Program().run();
  }
}
```

Figure 12. ArchJava code for the `Taprats` component. The main application method creates a `Program` component and invokes `run` on it. The initial `TileSelector` window is created in the field initializer for `ts`, and the `run` method wraps it in a `Frame`. The three private ports contain methods that create and connect new window components. Connect declarations show communication patterns between windows.

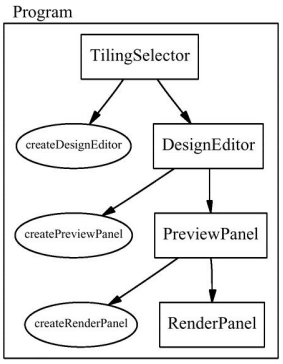


Figure 13. A visualization of the Taprats architecture, automatically derived from the ArchJava source code. Boxes represent subcomponents, and arrows represent inter-component control flow. The ovals are internal ports of the program component, which are used by the first three window components to create the next window in the sequence.

We discovered this connection by examining a violation of the Law of Demeter that forced us to restructure Taprats' code. After constructing a new window, the Taprats code called an accessor function to get a pane of that window, and then set the parameters of the pane's viewport directly—violating the Law of Demeter, since the pane was not an immediate neighbor of the original code. In our architecture, the pane was an internal component of the window component, and so this communication violated communication integrity. Therefore, we had to restructure the program to pass the viewport parameters to the enclosing window, which then passed them on to the pane. When shown the offending code, the developer agreed with our assessment and thought our solution was appropriate.

Despite this example, most of the Taprats code obeyed the Law of Demeter. This had a beneficial effect on our reengineering: when we converted an object into a component, the new component would often pass the compiler's communication integrity checks as soon as we converted direct method calls into calls on ports. Our experience suggests:

> **Hypothesis 5: It will be relatively easy to use ArchJava to express the software architecture of an object-oriented program whose source code obeys the Law of Demeter.**

Final Architecture. Figure 12 shows the ArchJava code that expresses the architecture of Taprats. The complete ArchJava source code for Taprats is available at the ArchJava web site [Arc02]. Compared to the developer's conceptual architecture, our final ArchJava architecture describes identical communication patterns between the user interface windows.

Figure 13 shows a visualization of the Taprats architecture automatically derived from the ArchJava source code using a visualization tool. We showed the developer this diagram, and he agreed that it captured his conceptual architecture well.

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 alternative Taprats architectures using ArchJava. For example, we could have followed the original source code more closely, producing a nested hierarchy of components instead of a linear sequence of components. Although this architecture would not show all of the user interface components and connections within one composite component, it would express the constraint that the user interface window instances form a tree with each window spawning multiple windows on the next level. The architecture we chose does not eliminate the possibility that the windows form a dag, where data from two source windows might be combined into a later-stage window (this does not occur in practice, of course). ArchJava is flexible enough to express both architectures, depending on which the software architect deems more appropriate.

5.3. Comparison to Aphyds Case Study

We found that expressing the conceptual architecture of Taprats with ArchJava was straightforward when compared with our earlier case study. In all, we spent approximately five times less effort in this case study than in the Aphyds case study, despite the fact that the programs were of similar size. Several application characteristics may have contributed to this difference:

- *Architecture Style.* The pipeline architecture style of Taprats, where data is passed from one component to another, has simpler communication patterns than the repository architecture style of Aphyds, where components access a shared database.
- *Architectural Connectivity.* Once spawned, Taprats' user interface windows are completely independent: they access different data, and do not communicate in any way. In contrast, Aphyds' user interface windows show different views of the same data, and therefore the user interface architecture includes connections to pass updated data and window state.
- *Architecture Granularity.* The developer of Aphyds specified a fairly fine-grained architecture, and the control flow within the user interface encouraged us to make the architecture even more fine-grained than the developer specified. In contrast, the Taprats user interface architecture was more coarse-grained, consisting of only four windows and their window panes.
- *Architectural Mismatches.* The structure of Taprats was quite similar to the architecture we tried to express. In the Aphyds study, we chose to make some previously dynamic structures static, requiring us to restructure the code to support re-initialization where new objects had been created previously.
- *Code Interdependence.* As described above, Taprats had a well-factored codebase that generally followed the Law of Demeter, making the architectural reengineering easy. In contrast, the Aphyds codebase contained many dependencies across object structures. Its frequent violations of the Law of

Demeter required many reengineering steps before the compiler's communication integrity checks were satisfied.

Our experience suggests that looking at these application characteristics may shed light on how much effort will be required to express an application's architecture with ArchJava.

5.4. Evaluation of the ArchJava Language

In general, our experience suggests that the ArchJava language design was adequate for expressing the architecture of Taprats. We were able to describe the conceptual architecture of the developer with minimal reengineering effort. The dynamic constructs of the language, which were largely untested in our earlier case study, were sufficient to express the dynamic nature of the Taprats user interface.

We also noticed areas in which the language design could be improved. As discussed before, it would be cleaner if each window in Taprats' user interface pipeline could create the next window in a more natural way, rather than requesting that the container component create the next window, as is done in the current solution. Also, in Figure 12, the creation ports (such as `createPreviewPanel`) that are connected to dynamically created child windows cannot accept an argument of component type telling them which window to connect, nor can they return a port with the correct type (as does the `createDesignEditor` port). Due to limitations in the current type system of ArchJava, two extra casts are required, one in the container component and one in the window component. We are considering ways to extend the ArchJava language design to handle these cases more smoothly.

5.5. Benefits of ArchJava

The ArchJava architecture has a number of advantages compared to the original, conceptual architecture of Taprats. 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 architectural visualizations can be generated automatically. Finally, it is easy to examine the source code to look at the interior structure of an ArchJava component, determine what methods are in each port, or examine how the methods are implemented.

The process of reengineering Taprats to make its architecture explicit may also have made the code more maintainable and easier to change. For example, the compiler's communication integrity checks identified several violations of the Law of Demeter, enabling us to replace them with better-factored code. Because ports encapsulate all control-flow communication between components, the components are more loosely coupled in the final version of the code, making them easier to evolve as requirements change. More experience with evolving ArchJava programs is needed to determine if these potential benefits are realized in practice.

In summary, we were able to capture the conceptual architecture of Taprats effectively in ArchJava with a small amount of effort relative to the size of the program. Our experience demonstrates that the language is flexible enough to describe dynamically evolving software architectures, and suggests future improvements to the language design.

6. 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 ensures that the implementation conforms to the specified architecture. Our formalization of ArchJava gives us confidence in its type system's ability to enforce communication integrity. A case study suggests that ArchJava can be applied with relatively little effort to moderate-sized Java programs with dynamically evolving architectures, making the program's structure explicit and improving the maintainability of code. 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.

Acknowledgements

We would like to thank David Garlan, Sorin Lerner, Vassily Litvinov, Vibha Sazawal, Todd Millstein, Matthai Philipose, and the anonymous reviewers for their comments and suggestions. We especially thank Craig Kaplan for his time and the Taprats 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.

References

[ACN02] Jonathan Aldrich, Craig Chambers, and David Notkin. ArchJava: Connecting Software Architecture to Implementation. Proc. International Conference on Software Engineering, Orlando, Florida, May 2002.

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

[Arc02] ArchJava web site. http://www.archjava.org/

[AL02] Andrei Alexandrescu and Konrad Lorincz. ArchJava: An Evaluation. University of Washington CSE 503 class report, available at http://www.archjava.org/, February 2002.

[BS98] Boris Bokowski and André Spiegel. Barat—A Front-End for Java. Freie Universität Berlin Technical Report B-98-09, December 1998.

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

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

[GHJ+94] Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, 1994.

[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 Igarashi, Benjamin Pierce, and Philip Wadler. Featherweight Java: A Minimal Core Calculus for Java and GJ. Proc. Object Oriented Programming Systems, Languages and Applications, Denver, Colorado, November 1999.

[Kap00] Craig S. Kaplan. Computer Generated Islamic Star Patterns. Proc. Bridges 2000: Mathematical Connections in Art, Music and Science, Winfield, Kansas, July 2000.

[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, Florida, October 2001.

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