Architectural Reasoning in ArchJava

Jonathan Aldrich Craig Chambers

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

{jonal, chambers}@cs.washington.edu

Abstract

Software architecture is a crucial part of the specification of component-based systems. Reasoning about software architecture can aid design, program understanding, and formal analysis. However, existing approaches decouple implementation code from architecture, allowing inconsistencies, causing confusion, violating architectural properties, and inhibiting software evolution. ArchJava is an extension to Java that seamlessly unifies a software architecture with its implementation. ArchJava's type system ensures that the implementation conforms to the architectural constraints. Therefore, programmers can visualize, analyze, reason about, and evolve architectures with confidence that architectural properties are preserved by the implementation.

1. Introduction

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

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

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

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

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

The rest of this paper is organized as follows. After the next section's discussion of related work, section 3 introduces the ArchJava language. Section 4 formalizes ArchJava's type system and outlines a proof of soundness and communication integrity in ArchJava. Section 5 briefly describes our initial experience with ArchJava. Finally, section 6 concludes with a discussion of future work.

2. Related Work

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

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

The UML is an example of specification languages that support various kinds of structural specification. UML's class diagrams can show the relationships between classes, and UML's object diagrams show relationships between object instances. However, in most UML tools, these diagrams are only intended to show *some* of the ways in which classes and objects can interact—they cannot be used to argue that no other kinds of interaction are possible, and thus do not support communication integrity. Object hierarchies can be expressed using composition

relationships, but this relationship does not enforce communication integrity either, because elements of the composition can still interact with outside objects.

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

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

Advanced module systems such as MzScheme's Units [FF98] and ML's functors [MTH90] can be used to encapsulate components and to describe the static architecture of a system. The FoxNet project [B95] shows how functors can be used to build up a network stack architecture out of statically connected components. However, these systems do not guarantee communication integrity in the language; instead, programmers must follow a careful methodology to ensure that each module communicates only with the modules it is connected to in the architecture.

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

3. The ArchJava Language

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

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

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

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

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

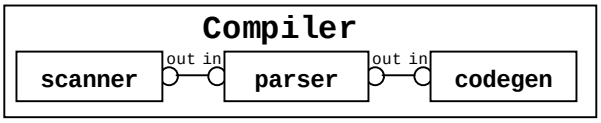
A prototype compiler for ArchJava is publicly available for download at the ArchJava web site [ACN01a]. Although in ArchJava the source code is the canonical representation of the architecture, visual representations are also important for conveying architectural structure. This paper uses hand-drawn diagrams to communicate architecture; however, we have also constructed a simple visualization tool that generates architectural diagrams automatically from ArchJava source code. In addition, we intend to provide an `archjavadoc` tool that would automatically construct graphical and textual web-based documentation for ArchJava architectures.

To allow programmers to describe software architecture, ArchJava adds new language constructs to support *components*, *connections*, and *ports*. The rest of this section describes by example how to use these constructs to express software architectures. Throughout the discussion, we show how the constructs work together to enforce communication integrity, culminating in a precise definition of communication integrity in ArchJava. Reports on the ArchJava web site [ACN01a] provide more information, including the complete language semantics and a formal proof of communication integrity in the core of ArchJava.

3.1 Components and Ports

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

A component instance communicates with external components through ports. A *port* represents a logical communication channel




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

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

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

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

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

between a component instance and one or more components that it is connected to.

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

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

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

3.2 Component Composition

In ArchJava, software architecture is expressed with *composite components*, which are made up of a number of subcomponents[1] connected together. Figure 2 shows how a compiler's architecture can be expressed in ArchJava. The example shows that the parser communicates with the scanner using one protocol, and with the code generator using another. The architecture also implies that the scanner does *not* communicate directly with the code generator. A primary goal of ArchJava is to ease program understanding tasks by supporting this kind of reasoning about program structure.

3.2.1 Subcomponents

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

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

3.2.2 Connections

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

Provided methods can be implemented by forwarding invocations to subcomponents or to the required methods of another port. The semantics of method forwarding and broadcast methods are given in the language reference manual on the ArchJava web site [ACN01a]. Alternative connection semantics, such as asynchronous communication, can be implemented in ArchJava by writing custom "smart connector" components that take the place of ordinary connections in the architecture.

3.3 Communication Integrity

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

Communication integrity in ArchJava means that components in an architecture can only call each others' methods along declared

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

connections between ports. Each component in the architecture can use its ports to communicate with the components to which it is connected. However, a component may not directly invoke the methods of components other than its children, because this communication may not be declared in the architecture—a violation of communication integrity. We define communication integrity more precisely in section 3.5.

3.4 Dynamic Architectures

The constructs described above express architecture as a static hierarchy of interacting component instances, which is sufficient for a large class of systems. However, some system architectures require creating and connecting together a dynamically determined number of components. Furthermore, even in programs with a static architecture, the top-level component must be instantiated at the beginning of the application.

3.4.1 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 `invoke` method. At creation time, each component records the component instance that created it as its *parent component*. For components like `Compiler` that are instantiated outside the scope of any component instance, the parent component is **`null`**.

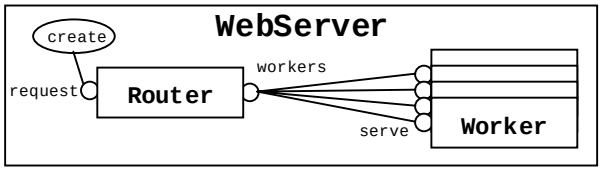
Communication integrity places restrictions on the ways in which component instances can be used. Because only a component's parent can invoke its methods directly, it is essential that typed references to subcomponents do not escape the scope of their parent component. This requirement is enforced by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

3.4.2 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`** `r.workers, Worker.serve` describes a set of connections between the component field `r` and dynamically created `Worker` components.

Each connect expression must match a connection pattern declared in the enclosing component. A connect expression *matches* a connection pattern if the connected ports are identical and each connected component instance is either the same component field specified in the pattern, or an instance of the type specified in the pattern. The connect expression in the web server example matches the corresponding connection pattern because




```
public component class WebServer {
  component Router r;
  connect r.request, create;
  connect pattern r.workers, Worker.serve;

  public void run() { r.listen(); }
  private port create {
    provides r.workers requestWorker() {
      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 = main.requestWorker();
      conn.httpRequest(sock.getInputStream(),
                       sock.getOutputStream());
    }
  }
}

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

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

the `newWorker` component in the connect expression is of static type `Worker`, the same type declared in the pattern.

3.4.3 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 at run time.

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 type allows access to. For example, in the `Router` component, the type **`this`**`.workers` refers to an instance of the `workers` port of the current `Router` component (in this case, **`this`** would be inferred automatically if it were omitted). The type `r.workers` refers to an instance of the `workers` port of the `r` subcomponent. This type can be used in method signatures such as `requestWorker` and local variable declarations such as `conn` in the `listen` method. Required methods can be invoked on expressions of port interface type, 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 `Worker.serve` and `r.workers`, and can therefore be assigned to a variable of type `r.workers`.

Provided methods can obtain the connection object through which the method call was invoked using the **`sender`** keyword. The detailed semantics of **`sender`** and other language features are covered in the ArchJava language reference available on the ArchJava web site [ACN01a].

3.4.4 Removing Components and Connections

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

3.5 Limitations of ArchJava

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

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

4. 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 and intuitively explains how it is enforced. Subsection 5.2 gives the static and dynamic semantics of ArchFJ, a language incorporating the core features of ArchJava. Finally, subsection 5.3 outlines proofs of communication integrity, subject reduction, and progress for ArchFJ.

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. Let the *execution scope* of component instance A on the run time stack, denoted *escope*(A), be any of A's executing methods and any of the object methods they transitively invoke, until another component's method is invoked.

Definition 1 [Dynamic Execution Scope]: Let `m` be an executing method with stack frame `mf`. If `m` is a component method, then `mf` ∈ *escope*(`this`). Otherwise, `mf` ∈ *escope*(*caller*(`mf`)).

Now we can define communication integrity:

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

1. `a` = `b`, *or*
2. `a` = *parent*(`b`), *or*
3. *parent*(`a`) = *parent*(`b`) ∧ "`connect [pattern] (f|t)`$_1$`.p`$_1$`,...,(f|t)`$_n$`.p`$_n$" ∈ *class*(*parent*(`a`))
   ∧ ∃i,j ∈ 1..n s.t. (*parent*(`a`).`f`$_i$ = `a` ∨ *type*(`a`)<:`t`$_i$) ∧
   (*parent*(`a`).`f`$_j$ = `b` ∨ *type*(`b`)<:`t`$_j$) ∧
   `m` ∈ *requiredmethods*(`p`$_i$) ∧
   `m` ∈ *providedmethods*(`p`$_j$)

Syntax:

```
CL      ::= class C extends C {C̄ f̄; K M̄}

CP      ::= component class P extends
            [P|Object] {C̄ f̄; K̄ M̄ R̄ X̄}

K       ::= E(C̄ f̄) {super(f̄); this.f̄ = f̄;}

M       ::= T m(T̄ x̄) { return e; }

R       ::= required T m(T̄ x̄)

X       ::= connect pattern (P̄)

e       ::= x
        |   e.f
        |   e.m(ē,_this)
        |   new C(ē)
        |   new P(ē, <fresh>, e_parent)
        |   (C)e
        |   (e.P_R)e
        |   cast(this, P, e)
        |   connect(ē, this)
        |   error
```

Figure 4. ArchFJ Syntax

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—that is, show that certain classes of errors cannot occur at run time. Unfortunately, proofs of type safety in a language like Java are extremely tedious due to the many cases involved, and to our knowledge the full Java language has never been formalized and proven type safe. Therefore, 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 modified Featherweight Java (FJ) to capture the essence of ArchJava in ArchFJ. ArchFJ makes a number of simplifications relative to ArchJava. ArchFJ leaves out ports; instead, each component class has a set of required and provided methods. Static connections and component fields are left out, as they are subsumed by dynamically created connections components. We also omit the **`sender`** keyword and broadcast methods. As in Featherweight Java (FJ), we omit interfaces. These changes make our type soundness proof shorter, but do not materially affect it otherwise.

4.2.1 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; `S`, `T`, and `V` range over types; `P` and `Q` range over component classes; `f` and `g` range over fields; `d` and `e` range over expressions; `l` ranges over labels generated by `<fresh>`; 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 includes the features of FJ plus a few extensions. Regular classes extend another class (which can be `Object`, a predefined class) and define a constructor `K` and a set of fields $\bar{f}$ and methods $\bar{M}$. Component classes can extend another component

Types:

```
T       ::= P
        | e.P_R
        | E
        | U(e.P̄_R)
```

Subtyping:

$$\frac{}{T <: T} \quad \text{(S-REFLEX)}$$

$$\frac{S <: T \qquad T <: V}{S <: V} \quad \text{(S-TRANS)}$$

$$\frac{P <: Q}{e_1.P_R <: e_2.Q_R} \quad \text{(S-REQUIRED)}$$

$$\frac{}{T <: \texttt{Object}} \quad \text{(S-OBJECT)}$$

$$\frac{e.P_R \in \overline{e.P_R}}{U(\overline{e.P_R}) <: e.P_R} \quad \text{(S-UNION)}$$

$$\frac{CT(\texttt{E}) = \texttt{[component] class E extends F \{ ... \}}}{\texttt{E} <: \texttt{F}} \quad \text{(S-EXTENDS)}$$

Figure 5. ArchFJ Types and Subtyping Rules

class, or `Object` (as in FJ, there are no interfaces). Component classes also declare a set of required methods $\bar{R}$ and a set of connection patterns $\bar{X}$ between their subcomponents.

Expressions include field lookup, method calls, object and component creation, various casts, a connect expression, and an error expression. These are extended from FJ in a few small ways:

- All method calls capture the current object `this` in an additional psuedo-argument which comes last and is not passed on to the callee.
- Components are labeled with a fresh label when they are created (labels in a method body are freshly generated when a method call is replaced with the method's body). This label allows us to reason about object identity in an otherwise functional language (assignment is not relevant to our type system or definition of communication integrity). Components also keep track of their parent, and which of their parent's component fields they were created with.
- In addition to regular casts to a class type, there are two new cast forms: one that allows casting to the required interface of a component (i.e., the set of methods the component requires), and another that allows casting to a component field type. The first cast accepts an instance expression type, while the latter cast includes an argument that captures the value of `this` in the current scope. Both arguments are used to verify the casts in the dynamic semantics.
- A connect expression conceptually creates a connection object on which components can invoke their required methods. The connect expression captures `this`, the parent object that created the connection.

Computation:

$$\frac{\mathit{fields}(\mathtt{E}) = \overline{\mathtt{C}}\ \overline{\mathtt{f}}}{(\mathtt{new\ E}(\overline{\mathtt{e}},\ldots)).\mathtt{f}_i \rightarrow \mathtt{e}_i} \quad \text{(R-FIELD)}$$

$$\frac{\mathit{mbody}(\mathtt{m},\mathtt{C}) = (\overline{\mathtt{x}}, \mathtt{e}_0)}{(\mathtt{new\ C}(\overline{\mathtt{e}})).\mathtt{m}(\overline{\mathtt{d}},\ \mathtt{d}_{\mathtt{this}}) \rightarrow [\overline{\mathtt{d}}/\overline{\mathtt{x}},\ \mathtt{new\ C}(\overline{\mathtt{e}})/\mathtt{this}]\mathtt{e}_0} \quad \text{(R-INVK)}$$

$$\frac{\mathtt{E} <: \mathtt{C}}{(\mathtt{C})(\mathtt{new\ E}(\ldots)) \rightarrow \mathtt{new\ E}(\ldots)} \quad \text{(R-CAST)}$$

$$\frac{\mathtt{e} = \mathtt{new\ E}(\ldots) \not<: \mathtt{C} \ \vee\ \mathtt{e} = \mathtt{connect}(\ldots)}{(\mathtt{C})(\mathtt{e}) \rightarrow \mathtt{error}} \quad \text{(E-CAST)}$$

$$\frac{\begin{array}{c}\mathtt{e} = \mathtt{new\ P}(\overline{\mathtt{e}},\ \mathtt{l},\ \mathtt{e}_{\mathtt{parent}}) \\ \mathit{mbody}(\mathtt{m},\mathtt{P}) = (\overline{\mathtt{x}}, \mathtt{e}_0) \\ \mathtt{d}_{\mathtt{this}} = \mathtt{e} \vee \mathtt{d}_{\mathtt{this}} = \mathtt{e}_{\mathtt{parent}}\end{array}}{\mathtt{e}.\mathtt{m}(\overline{\mathtt{d}},\ \mathtt{d}_{\mathtt{this}}) \rightarrow [\overline{\mathtt{d}}/\overline{\mathtt{x}},\ \mathtt{e}/\mathtt{this}]\mathtt{e}_0} \quad \text{(R-PINVK)}$$

$$\frac{\mathtt{e} = \mathtt{new\ P}(\overline{\mathtt{e}},\ \mathtt{l},\ \mathtt{e}_{\mathtt{this}}) \qquad \mathtt{P} <: \mathtt{Q}}{\mathtt{cast}(\mathtt{e}_{\mathtt{this}},\ \mathtt{Q},\ \mathtt{e}) \rightarrow \mathtt{e}} \quad \text{(R-PCAST)}$$

$$\frac{\begin{array}{c}\mathtt{e} = \mathtt{connect}(\ldots) \ \vee \\ (\mathtt{e} = \mathtt{new\ P}(\overline{\mathtt{e}},\ \mathtt{l},\ \mathtt{e}_{\mathtt{parent}}) \\ \mathtt{where\ e}_{\mathtt{this}} \neq \mathtt{e}_{\mathtt{parent}} \vee \mathtt{P} \not<: \mathtt{Q})\end{array}}{\mathtt{cast}(\mathtt{e}_{\mathtt{this}},\ \mathtt{Q},\ \mathtt{e}) \rightarrow \mathtt{error}} \quad \text{(E-PCAST)}$$

$$\frac{\mathtt{e}_{\mathtt{cast}} = \mathtt{new\ P}(\ldots) \qquad \mathtt{e}_{\mathtt{cast}} \in \overline{\mathtt{e}} \qquad \mathtt{P} <: \mathtt{Q}}{\begin{array}{c}(\mathtt{e}_{\mathtt{cast}}.\mathtt{Q}_{\mathtt{R}})(\mathtt{connect}(\overline{\mathtt{e}}, \mathtt{e}_{\mathtt{this}})) \\ \rightarrow \mathtt{connect}(\overline{\mathtt{e}}, \mathtt{e}_{\mathtt{this}})\end{array}} \quad \text{(R-RCAST)}$$

$$\frac{\begin{array}{c}\mathtt{e} = \mathtt{new\ E}(\ldots) \ \vee \\ \mathtt{e} = \mathtt{connect}(\overline{\mathtt{e}}, \mathtt{e}_{\mathtt{this}}) \\ \mathtt{where\ e}_{\mathtt{cast}} \notin \overline{\mathtt{e}} \ \vee\ \mathtt{e}_{\mathtt{cast}} = \mathtt{new\ F}(\ldots) \not<: \mathtt{Q}\end{array}}{(\mathtt{e}_{\mathtt{cast}}.\mathtt{Q}_{\mathtt{R}})(\mathtt{e}) \rightarrow \mathtt{error}} \quad \text{(E-RCAST)}$$

$$\frac{\begin{array}{c}\mathtt{d}_{\mathtt{this}} \in \overline{\mathtt{e}} \qquad \mathit{legal}(\mathtt{connect}(\overline{\mathtt{e}}, \mathtt{e}_{\mathtt{this}})) \\ \mathit{mbody}(\mathtt{m}, \mathtt{connect}(\overline{\mathtt{e}}, \mathtt{e}_{\mathtt{this}})) = (\overline{\mathtt{x}}, \mathtt{e}_0, \mathtt{e}_i)\end{array}}{\begin{array}{c}(\mathtt{connect}(\overline{\mathtt{e}}, \mathtt{e}_{\mathtt{this}})).\mathtt{m}(\overline{\mathtt{d}},\ \mathtt{d}_{\mathtt{this}}) \\ \rightarrow [\overline{\mathtt{d}}/\overline{\mathtt{x}},\ \mathtt{e}_i/\mathtt{this}]\mathtt{e}_0\end{array}} \quad \text{(R-XINVK)}$$

Figure 6. ArchFJ Reduction Rules

- We represent failed dynamic checks (such as casts) with an explicit `error` value, to make our progress theorem cleaner to state.

4.2.2 Types and Subtypes

ArchJava's types and subtyping rules are given in Figure 5. Types include class and component types (`E`), required interface types of components ($\mathtt{e}.\mathtt{P}_{\mathtt{R}}$), and union types of multiple required interfaces. Subtyping of classes and components is defined by the reflexive, transitive closure of the immediate subclass relation given by the `extends` clauses in *CT*. We require that there are no cycles in the induced subtype relation. Required interface types follow the subtyping relation of components (ignoring the instance expressions, which are reasoned about separately from subtyping). Finally, every type is a subtype of `Object`, and a union is a subtype of all its member types.

4.2.3 Reduction Rules

The reduction relation, defined by the reduction rules given in Figure 6, is of the form $\mathtt{e} \rightarrow \mathtt{e}'$, read "expression `e` reduces to expression `e'` in one step." We write $\rightarrow^*$ for the reflexive, transitive closure of $\rightarrow$. The only unusual reduction rule is R-XINVK, which allows method invocation on connection expressions. The *mbody* helper function does a lookup to determine the correct method body to invoke. Two error rules are defined representing casts that are not guaranteed to succeed by the type system presented below. The reduction rules can be applied at any point in an expression, so we also need appropriate congruence rules (such as if $\mathtt{e} \rightarrow \mathtt{e}'$ then $\mathtt{e.f} \rightarrow \mathtt{e}'\mathtt{.f}$), which we omit here. Furthermore, we assume an order of evaluation that follows Java's normal evaluation rules.

4.2.4 Typing Rules

Most of the typing rules given in Figure 7 are standard. Typing judgments are given in an environment Γ, a finite mapping from variables to types. Rule T-INVK places constraints on passing connection objects to an argument position declared with a required interface and instance expression of **`this`**, to ensure that the connection object does indeed connect the receiver object. Rule T-PNEW introduces qualified component types. Rule T-CONNECT introduces union types for connections. In addition, T-CONNECT verifies that some connection pattern in the current component matches the types of the connected objects; this will be important later for establishing that reduction cannot get stuck due to an illegal connection.

Class, method, and connection typing rules check for well-formed class definitions, and have the form "class declaration `E` is OK," and "method/connection `X` is OK in `E`." The rules for class and method typing are similar to those in FJ. In the case of component classes, the typing rule verifies that only subclasses of `Object` may define required methods—as in ArchJava, component subclasses may only inherit existing required methods from their component superclass. The connection typing rule verifies that each required method has a unique provided method with the right signature, and that every method name has only one signature across all the required methods.

We have made one significant simplification relative to FJ. We do not distinguish between upcasts, downcasts, and so-called "stupid casts" which cast one type to an unrelated one. This means that our type system does not check for "stupid casts" in the original typing derivation, as Java's type system does. However, the change shortens our presentation and proofs considerably, and the stupid casts technique from FJ can be easily applied to our system to get the same checks that are present in Java.

Expression Typing:

$$\Gamma \vdash x \in \Gamma(x) \qquad \text{(T-VAR)}$$

$$\frac{\Gamma \vdash e_0 \in C_0 \qquad \mathit{fields}(C_0) = \overline{C}\ \overline{f}}{\Gamma \vdash e_0.f_i \in C_i} \qquad \text{(T-FIELD)}$$

$$\frac{\begin{array}{c}\Gamma \vdash e_0 \in T_0 \qquad \mathit{mtype}(m, T_0[, e_{this}]) = \overline{T} \to T \\ \Gamma \vdash \overline{e} \in \overline{S} \qquad \overline{S} <: \overline{T} \\ \Gamma \vdash e_{this} \in T_{this} \\ T_i = \texttt{this}.P_{Ri}\ \text{implies}\ S_i = e_0.S_{Ri}\end{array}}{\Gamma \vdash e_0.m(\overline{e}, e_{this}) \in T[e_0/\texttt{this}]} \qquad \text{(T-INVK)}$$

$$\frac{\mathit{fields}(C) = \overline{D}\ \overline{f} \qquad \Gamma \vdash \overline{e} \in \overline{C} \qquad \overline{C} <: \overline{D}}{\Gamma \vdash \texttt{new}\ C(\overline{e}) \in C} \qquad \text{(T-NEW)}$$

$$\frac{\begin{array}{c}\mathit{fields}(P) = \overline{D}\ \overline{f} \qquad \Gamma \vdash \overline{e} \in \overline{C} \qquad \overline{C} <: \overline{D} \\ \Gamma \vdash e_p \in T_p\end{array}}{\Gamma \vdash \texttt{new}\ P(\overline{e}, \langle \texttt{fresh} \rangle, e_p) \in P} \qquad \text{(T-PNEW)}$$

$$\frac{\begin{array}{c}\Gamma \vdash e_{this} \in P_{this} \qquad \Gamma \vdash \overline{e} \in \overline{P} \qquad \overline{P} <: \overline{Q} \\ \texttt{connect pattern}(\overline{Q}) \in \mathit{connects}(P_{this})\end{array}}{\Gamma \vdash \texttt{connect}(\overline{e}, e_{this}) \in U(\overline{e.P_R})} \qquad \text{(T-CONNECT)}$$

$$\frac{\Gamma \vdash e \in T_0}{\Gamma \vdash ([e_{cast}.P_R | C])e \in [e_{cast}.P_R | C]} \qquad \text{(T-CAST)}$$

$$\frac{\Gamma \vdash e \in T \qquad \Gamma \vdash e_{this} \in P_{this}}{\Gamma \vdash \texttt{cast}(e_{this}, Q, e) \in Q} \qquad \text{(T-PCAST)}$$

Class Typing:

$$\frac{\begin{array}{c}K = F(\overline{D}\ \overline{g}, \overline{C}\ \overline{f})\{\texttt{super}(\overline{g}); \texttt{this}.\overline{f} = \overline{f};\} \\ \mathit{fields}(E) = \overline{D}\ \overline{g} \qquad \overline{M}[, \overline{X}]\ \texttt{OK IN}\ P \\ E = \texttt{Object} \vee \#(\overline{R}) = 0\end{array}}{\begin{array}{c}[\texttt{component}]\ \texttt{class}\ F\ \texttt{extends}\ E \\ \{\overline{C}\ \overline{f};\ K\ \overline{M}\ [\overline{R}\ \overline{X}]\}\ \texttt{OK}\end{array}} \qquad \text{(T-CLASS)}$$

Method Typing:

$$\frac{\begin{array}{c}\overline{x}:\overline{T}, \texttt{this}:E \vdash e \in S \qquad S <: T \\ CT(E) = [\texttt{component}]\ \texttt{class}\ E\ \texttt{extends}\ F\ \{..\} \\ \mathit{override}(m, F, \overline{T} \to T) \qquad T, \overline{T}\ \text{not components}\end{array}}{T\ m(\overline{T}\ \overline{x})\{\ \texttt{return}\ e;\}\ \texttt{OK in}\ E} \qquad \text{(T-METH)}$$

Connection Typing:

$$\frac{\begin{array}{c}\forall i\ \ \mathit{mtype}(m, P_{Ri}) = \overline{T} \to T\ \text{ implies} \\ \exists j \neq i\ \ s.t.\ \mathit{mtype}(m, P_j) = \overline{T} \to T \\ \wedge\ \forall k \neq j\ \ \mathit{mtype}(m, P_k)\ \ \mathit{not\ \ defined} \\ \forall i, j\ \ \mathit{mtype}(m, P_{Ri}) = \overline{T} \to T \wedge \mathit{mtype}(m, P_{Rj}) = \overline{S} \to S \\ \text{implies}\ \ \overline{T} = \overline{S} \wedge T = S\end{array}}{\texttt{connect pattern}(\overline{P})\ \texttt{OK IN}\ Q} \qquad \text{(T-X)}$$

Figure 7. ArchFJ Static Semantics

Field lookup:

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

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

Connection lookup:

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

$$\frac{\begin{array}{c}CT(P) = \texttt{component class}\ P\ \texttt{extends}\ E \\ \{\overline{C}\ \overline{f};\ K\ \overline{M}\ \overline{R}\ \overline{X}\} \\ \mathit{connects}(E) = \overline{X_0}\end{array}}{\mathit{connects}(P) = \overline{X_0}, \overline{X}}$$

Method type lookup:

$$\frac{\begin{array}{c}CT(E) = [\texttt{component}]\ \texttt{class}\ E\ \texttt{extends}\ F\ \{..\overline{M}..\} \\ T\ m\ (\overline{T}\ \overline{x})\ \{\texttt{return}\ e;\} \in \overline{M}\end{array}}{\mathit{mtype}(m, E) = \overline{T} \to T}$$

$$\frac{\begin{array}{c}CT(E) = [\texttt{component}]\ \texttt{class}\ E\ \texttt{extends}\ F\ \{..\overline{M}..\} \\ m\ \text{ is not defined in }\ \overline{M}\end{array}}{\mathit{mtype}(m, E) = \mathit{mtype}(m, F)}$$

$$\frac{\begin{array}{c}CT(P) = \texttt{component class}\ P\ \texttt{extends}\ E\ \{..\overline{R}..\} \\ \texttt{required}\ T\ m\ (\overline{T}\ \overline{x}) \in \overline{R}\end{array}}{\mathit{mtype}(m, P) = \overline{T} \to T}$$

$$\frac{\begin{array}{c}CT(P) = \texttt{component class}\ P\ \texttt{extends}\ E\ \{..\overline{R}..\} \\ m\ \text{ is not declared in }\ \overline{R}\end{array}}{\mathit{mtype}(m, e.P_R) = \mathit{mtype}(m, e.E_R)}$$

$$\frac{e_{this} = e_i \qquad \mathit{mtype}(m, e_i.P_{Ri}) = \overline{T} \to T}{\mathit{mtype}(m, U(\overline{e.P_R}), e_{this}) = \overline{T} \to T}$$

Figure 8. ArchFJ Auxiliary Definitions

4.2.5 Auxiliary Definitions

Most of the auxiliary definitions shown in Figures 8 and 9 are straightforward and are taken from FJ. The connection typing rule verifies that the passed-in **`this`** expression is one of the instance expressions in the union type. The connection method lookup rule chooses the component *i* providing the method with *mtype*, based on the static types in the original connection declaration. It is guaranteed to choose a unique component because the connection typing rule implies that *mtype* is only defined for one of the types in the connection. It then picks the actual method body dynamically using the usual *mbody* rule. Finally, it returns the expression to be passed as **`this`** in the method call.

The *legal* rule checks that a connect expression corresponds to a connection pattern. It also verifies that the connect expression was created inside the parent component of each sibling.

Method body lookup:

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

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

$$\frac{\mathtt{e_{this}} = \mathtt{new}\ \mathtt{P_{this}}(\ldots) \qquad \overline{\mathtt{e}} = \mathtt{new}\ \overline{\mathtt{Q}}(\ldots) \qquad \mathtt{connect\ pattern}(\overline{\mathtt{P}}) \in connects(\mathtt{P_{this}}) \qquad \overline{\mathtt{Q}} <: \overline{\mathtt{P}} \qquad mtype(\mathtt{m}, \mathtt{P}_i) = \overline{\mathtt{T}} \to \mathtt{T} \qquad mbody(\mathtt{m}, \mathtt{Q}_i) = (\overline{\mathtt{x}}, \mathtt{e}_0)}{mbody(\mathtt{m}, \mathtt{connect}(\overline{\mathtt{e}}, \mathtt{e_{this}})) = (\overline{\mathtt{x}}, \mathtt{e}_0, \mathtt{e}_i)}$$

Legal Connections:

$$\frac{\mathtt{e_{this}} = \mathtt{new}\ \mathtt{P_{this}}(\ldots) \qquad \mathtt{e}_i = \mathtt{new}\ \mathtt{Q}_i(\overline{\mathtt{d}_i}, \mathtt{l}_i, \mathtt{e}_{pi}) \qquad \mathtt{connect\ pattern}(\overline{\mathtt{P}}) \in connects(\mathtt{P_{this}}) \qquad \overline{\mathtt{Q}} <: \overline{\mathtt{P}} \qquad \forall \mathtt{i}\ \mathtt{e}_i = \mathtt{e_{this}} \vee \mathtt{e}_{pi} = \mathtt{e_{this}}}{legal(\mathtt{connect}(\overline{\mathtt{e}}, \mathtt{e_{this}}))}$$

Valid method overriding:

$$\frac{mtype(\mathtt{m},\ \mathtt{E}) = \overline{\mathtt{T}} \to \mathtt{T}_0,\ \text{implies}\ \overline{\mathtt{S}} = \overline{\mathtt{T}}\ \text{and}\ \mathtt{S}_0 = \mathtt{T}_0}{override(\mathtt{m},\ \mathtt{E},\ \overline{\mathtt{S}} \to \mathtt{S}_0)}$$

Figure 9. More Auxiliary Definitions

4.3 Theorems

We state three main theorems: communication integrity, subject reduction, and progress. Subject reduction and progress together imply that the ArchJava type system is sound. First, the reduction rules ensure communication integrity:

Theorem [Communication Integrity in ArchFJ]:

1. For all direct method invocations on a component `P` that succeed, either `P` or `P`'s parent component is the current component `this`.
2. For all method invocations on a connection that succeed, the current component `P` is part of the connection, `P` and the component `Q` being invoked either have the same parent or one is the parent of the other, and the parent `P'` declared a connection pattern between `P` and `Q`.

Proof: Part 1 of communication integrity is ensured by the precondition $\mathtt{d_{this}} = \mathtt{e} \vee \mathtt{d_{this}} = \mathtt{e_{parent}}$ of R-PINVK. Part 2 of communication integrity is ensured by the precondition $\mathtt{d_{this}} \in \overline{\mathtt{e}}$ of R-XINVK as well as the definition of *legal*.

The presentation of our Subject Reduction and Progress theorems is adapted from FJ [IPW99].

Theorem [Subject Reduction]: If $\Gamma \vdash \mathtt{e}_0 \in \mathtt{T}_0$ and $\mathtt{e}_0 \to \mathtt{e}_1$, then $\Gamma \vdash \mathtt{e}_1 \in \mathtt{T}_1$ for some $\mathtt{T}_1 <: \mathtt{T}_0$.

Proof sketch: The main property required is the following term-substitution lemma:

Lemma 1 [Term Substitution]: If $\Gamma, \overline{\mathtt{x}}:\overline{\mathtt{S}_0} \vdash \mathtt{e} \in \mathtt{T}_0$ and $\Gamma \vdash \overline{\mathtt{d}} \in \overline{\mathtt{S}_1}$ where $\overline{\mathtt{S}_1} <: \overline{\mathtt{S}_0}$, then $\Gamma \vdash [\overline{\mathtt{d}}/\overline{\mathtt{x}}]\mathtt{e} \in \mathtt{T}_1$ for some $\mathtt{T}_1 <: \mathtt{T}_0$.

Lemma 1 is proved by induction on the derivation of $\Gamma, \overline{\mathtt{x}}:\overline{\mathtt{S}_0} \vdash \mathtt{e} \in \mathtt{T}_0$.

The theorem itself can then be proved by induction on the derivation of $\mathtt{e}_0 \to \mathtt{e}_1$, with a case analysis on the last rule used. Lemma 1 is useful in many of the steps, and especially for the congruence rules.

The only tricky case is to show that the preconditions of T-INVK still hold after a reduction step. This can be shown based on a case analysis on the introduction of required component types (T-INVK, T-CONNECT, and T-CAST), and a lemma that term substitution preserves the required relationships among instance expressions.

Theorem [Progress]: Suppose `e` is a well-typed expression. Then either `e` has an `error` subexpression, or `e` is a *value* made up of only new and connect expressions, or $\mathtt{e} \to \mathtt{e}'$.

Proof sketch: The theorem is proved by induction on the derivation of the reduction of `e`. For each reduction rule, we show that any valid typing for the subexpressions in the left-hand-side, together with the assumption of progress for the subexpression, implies the preconditions for the reduction rule. In most cases the implication is clear, but two interesting lemmas are necessary for rules R-PINVK and R-XINVK, respectively.

Lemma 2 [An expression of component type reduces to `this` or a direct child component of `this`]:

Consider an expression $\mathtt{e_t.m}(\overline{\mathtt{e}}, \ldots)$ where $\mathtt{e_t} = \mathtt{new\ E}(\ldots)$, $mbody(\mathtt{m}, \mathtt{E}) = (\mathtt{x}, \mathtt{e}_0)$, and $\mathtt{e}_0$ has a subexpression $\mathtt{e}_1\mathtt{.m}(\overline{\mathtt{e}_1}, \mathtt{this})$. If $\overline{\mathtt{x}}:\overline{\mathtt{T}}, \mathtt{this:E} \vdash \mathtt{e}_1 \in \mathtt{P}$ and $[\mathtt{d/x}, \mathtt{e_t/this}]\mathtt{e}_1 \to^* \mathtt{new\ Q}(\ldots, \mathtt{e_{parent}})$, then either $\mathtt{e}_1 = \mathtt{this}$ or $\mathtt{e_{parent}} \to^* \mathtt{e_t}$.

This lemma can be proved by a case analysis of the last typing rule used in the typing derivation of $\mathtt{e}_1$. There are only three rules that result in a component type: T-VAR, T-PNEW, and T-PCAST (methods cannot return component type, by the well-formed method rule). The T-VAR rule gives a component type to a variable `x`, but the only way a component type can be introduced into Γ is by the component method typing rule, with $\mathtt{x} = \mathtt{this}$. If the component type was introduced in T-PNEW, $\mathtt{e}_1 = \mathtt{new\ Q}(\ldots, \mathtt{this})$ and so $\mathtt{e_{parent}} = \mathtt{e_t}$. If the component type came from T-PCAST, $\mathtt{e}_1$ must be of the form $\mathtt{cast}(\mathtt{this}, \mathtt{P}, \mathtt{new\ Q}(\ldots, \mathtt{e_{parent}}))$, and so the derivation of $[\mathtt{d/x}, \mathtt{e_t/this}]\mathtt{e}_1 \to^* \mathtt{new\ Q}(\ldots, \mathtt{e_{parent}})$ must include a reduction rule R-PCAST which verifies that $\mathtt{e_{parent}} = \mathtt{e_t}$ in the final expression.

Lemma 3 [Well-typed connection expressions are legal]: If $\Gamma \vdash \mathtt{connect(e, e_{this})} \in \mathtt{T}$ then $legal(\mathtt{connect(e, e_{this})})$.

The typing rule T-CONNECT, together with Lemma 2, demonstrates that all the required properties in *legal* hold.

5. Evaluation

We have written a prototype compiler for ArchJava, which is available for download from the ArchJava web site [ACN01a]. In order to determine whether the ArchJava language enables effective component-oriented programming, we undertook a case study applying ArchJava to Aphyds, a 12,000-line circuit design application written in Java.

Results from our case study [ACN01b] indicate that for this program, the developer's architecture can be expressed in ArchJava with relatively little effort (about 30 programmer hours). The resulting architecture yields insight into the program's communication patterns, and may be useful in eliminating software defects.

6. Conclusion and Future Work

ArchJava allows programmers to effectively express software architecture and then seamlessly fill in the implementation with Java code. This paper has motivated and outlined a language design integrating architecture and implementation, and proved type soundness and communication integrity in a formalization of ArchJava. At every stage of development and evolution, ArchJava enforces communication integrity, ensuring that the implementation conforms to the specified architecture. Thus, ArchJava helps to promote effective architecture-based design, implementation, program understanding, and evolution.

In future work, we intend to extend the case study to larger programs, to see if ArchJava can be successfully applied to programs of 100,000 lines and up. We will also investigate extending the language design to enable more advanced reasoning about component-based systems, including temporal ordering constraints on component method invocations and constraints on data sharing between components.

7. Acknowledgements

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

8. References

[ACN01a] Jonathan Aldrich, Craig Chambers, and David Notkin. ArchJava web site. http://www.cs.washington.edu/homes/jonal/archjava/

[ACN01b] Jonathan Aldrich, Craig Chambers, and David Notkin. Component-Oriented Programming in ArchJava. In Proceedings of the OOPSLA '01 Workshop on Language Mechanisms for Programming Software Components, July 2001. Available at http://www.cs.washington.edu/homes/jonal/archjava/

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

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

[FF98] M. Flatt and M. Felleisen. Units: Cool modules for HOT languages. In PLDI'98 - ACM Conf. on Programming Language Design and Implementation, pages 236--248, 1998.

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

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

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

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

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

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

[MT00] Nenad Medvidovic and Richard N. Taylor. A Classification and Comparison Framework for Software Architecture Description Languages. *IEEE Transactions on Software Engineering*, vol. 26, no. 1, pp. 70-93, January 2000.

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

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

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

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

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