Using Types to Enforce Architectural Structure

Jonathan Aldrich

A dissertation submitted in partial fulfillment of the
requirements for the degree of

Doctor of Philosophy

University of Washington

2003

Program Authorized to Offer Degree: Computer Science and Engineering

University of Washington
Graduate School

This is to certify that I have examined this copy of a doctoral dissertation by

Jonathan Aldrich

and have found that it is complete and satisfactory in all respects,
and that any and all revisions required by the final
examining committee have been made.

Chairs of Supervisory Committee:

_____
Craig Chambers

_____
David Notkin

Reading Committee:

_____
Craig Chambers

_____
David Notkin

_____
Alan Borning

Date: _____

In presenting this dissertation in partial fulfillment of the requirements for the Doctoral degree at the University of Washington, I agree that the Library shall make its copies freely available for inspection. I further agree that extensive copying of the dissertation is allowable only for scholarly purposes, consistent with "fair use" as prescribed in the U.S. Copyright Law. Requests for copying or reproduction of this dissertation may be referred to Bell and Howell Information and Learning, 300 North Zeeb Road, P.O. Box 1346, Ann Arbor, MI 48106-1346, to whom the author has granted "the right to reproduce and sell (a) copies of the manuscript in microform and/or (b) printed copies of the manuscript made from microform."

Signature: _____

Date: _____

University of Washington

Abstract

Using Types to Enforce Architectural Structure

Jonathan Aldrich

Chairs of the Supervisory Committee:

Professor Craig Chambers

Professor David Notkin

Computer Science and Engineering

Software architecture describes the high-level structure of a software system, and can be used for design, analysis, and software evolution tasks. However, existing tools decouple architecture from implementation, allowing inconsistencies to accumulate as a software system evolves. Because of the potential for inconsistency, engineers evolving a program cannot fully trust the architecture to accurately describe the properties or structure of the implementation.

This dissertation explores a new approach: integrating architectural descriptions into an implementation language, and using a type system to ensure that the architectural structure is consistent with the code. The approach is embodied in the ArchJava language, which extends Java with features that document the software architecture and data sharing within a system. ArchJava's type system enforces communication integrity, the property that implementation components communicate only along connections declared in the architecture. ArchJava is flexible enough to describe architectures that may change at run time, and it supports many of the same coding styles and idioms that programmers use in Java. Several case studies applying ArchJava to existing programs of significant size provide preliminary evidence that ArchJava is practical and can aid software evolution tasks.

TABLE OF CONTENTS

LIST OF FIGURES

iv

ACKNOWLEDGEMENTS

First, I would like to thank my wife, Becky Billock, for her incredible support throughout my graduate career. She has been amazingly understanding and supportive when I am swamped with paper deadlines. She has provided me with encouragement when I am down, a sounding board when I am pensive, and joy all of the time. I will be forever grateful.

I thank my parents for fostering a love of learning from day one, and always encouraging me to put forth my best. I appreciate my mom for reading to me from the very beginning. I thank my dad for always being ready to answer my scientific questions. They gave me the best start anyone could have.

A huge thank you goes to my advisors, Craig Chambers and David Notkin. Looking back, I cannot believe how much they have taught me over the last six years. They have provided great advice not only on technical issues, but also on the intangible aspects of maturing as a researcher. I appreciate the encouragement and feedback from the other members of my committee, Alan Borning, Gaetano Borriello, and Blake Hannaford. I also want to thank those who mentored me in computer science before graduate school: Ron Tenison, Rusty Whitney, Lougie Anderson, Mani Chandy, and others.

I would like to thank the many other people who have provided helpful comments on various parts of this dissertation: Doug Lea, David Garlan, Todd Millstein, Sorin Lerner, Keunwoo Lee, Vassily Litvinov, Vibha Sazawal, Matthai Philipose, Anthony LaMarca, Stefan Sigurdsson, Matt Lease, Andrei Alexandrescu, and many anonymous reviewers. Without their suggestions, this work could not have been as strong. I would also like to thank Scott Hauck, Craig Kaplan, and the researchers at Intel Research Seattle for their time and for access to the programs used in my case studies.

I would like to thank Craig Chambers, David Notkin, David Garlan, Doug Lea, and Susan Eggers, who spent hours writing recommendation letters for me. I also thank the many people I interacted with during my job search—it was a wonderful experience getting to know new people and getting exposure to new ideas. Finally, I would like to thank the faculty at Carnegie Mellon University for giving me a job.

Chapter 1

Introduction

Building and evolving large software systems is the biggest challenge facing software engineers today [Par72, LB85]. The sheer size and complexity of application and system software—millions of lines of code in many cases—means that no single engineer is knowledgeable enough to confidently change every part of it. Thus, each engineer must focus on one or more components of the system. In order to effectively evolve a component, however, an engineer must often understand how that component interacts with other parts of the system. For example, when modifying the invariants of a data structure, an engineer must discover what code within and outside the component relies on those invariants, and make appropriate modifications to that code. Understanding how components interact is especially difficult in many modern systems, which communicate indirectly through shared data structures, dynamic dispatch, and events. To evolve these programs effectively, an engineer often needs an abstraction of the possible run-time types and aliases of each object involved in the change. Such abstractions are difficult to gain, and if they are incorrect, engineers are likely to inject defects as they evolve the software system.

System scale poses challenges not only to correctness, but also to a broad range of other software development issues. For example, careful engineering techniques can largely eliminate security holes from a small software system. However, it is extremely challenging to eliminate these errors from a large system, because it is difficult to analyze how control and data may flow between secure and untrusted parts of the system. Similarly, it can be difficult to isolate the cause of a performance problem in a large program because of the complex ways in which system components interact. Although there is no panacea for solving these problems, a common underlying issue is the need to understand and control how different parts of a software system communicate.

1.1 Software Architecture

For years, software architects have used informal, high-level design models as conceptual tools for managing the complexity of large software systems. Whether drawn informally on a white board, or included in more formal design documentation, these models describe the high-level interactions between different components in a software system. The intent of these models is to communicate to an entire engineering team part of the global knowledge needed to develop and evolve each component of the system.

In the last decade, the discipline of software architecture has provided a formal semantic framework and tools for reasoning about these high-level design models [PW92, GS93]. Software architecture is the high-

level 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. For example, an architectural model can be analyzed to prove the absence of deadlock [AG97], or to determine whether secure information could flow to an untrusted component. Software architecture can also facilitate the implementation and evolution of large software systems. For instance, 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 come with tools that generate code to connect independently developed components [SDK+95,LV95]. However, there is no guarantee that the components themselves obey the architectural constraints of the system. Instead, these ADLs depend on component developers to follow style guidelines that prohibit common programming idioms such as data sharing. Other ADLs are purely for modeling and analysis, requiring programmers to implement the system without any automated support [AG97,MQR95]. Thus, it may be difficult to trace architectural features to the implementation. None of these systems guarantees consistency between architecture and implementation, so while the architecture may be analyzed for certain properties, there is no way to know if the properties hold in the implementation.

Even if a system is initially built to conform to its intended architecture, as the system evolves to address new requirements, its implementation may become inconsistent with the original architecture over time. This inconsistency causes problems for engineers working with the system, making it difficult to understand parts of the system in isolation, and causing program errors when engineers rely on their inaccurate architectural models. In summary, inconsistency between architecture and implementation pervades existing systems, causing problems both in human reasoning and automated analysis of programs.

1.2   Communication Integrity

In order to enable architectural reasoning about an implementation, the implementation must conform to its architecture. A system conforms to its architecture if the architecture is a correct abstraction of the run-time behavior of the system. The *communication integrity* property defines how architectural structure constrains run-time communication in the implementation [MQR95,LV95]:

> **Definition [Communication integrity]:** Each component in the implementation may only communicate directly with the components to which it is connected in the architecture.

Enforcing communication integrity is challenging due to programming language mechanisms that obscure communication pathways, including references, objects, and first-class functions. Previous systems have made serious compromises in order to enforce communication integrity. For example, analysis-based systems have been limited to fairly simple architectural models in order to make the analysis scalable [MNS01,LR03].

Systems can enforce communication integrity by restricting the implementation language to prohibit pointers and other problematic features. For example, SDL is a domain-specific language for real-time telecommunications applications, providing a structured language without pointers or dynamic dispatch [ITU99]. However, shared data is an important and widely recognized form of inter-component communication, and forbidding it entirely is unrealistic in the context of general-purpose languages.

Luckham and Vera suggest that developers follow style guidelines (such as avoiding shared data) in order to preserve communication integrity [LV95]. Not only do these guidelines place significant restrictions on implementers, there is no guarantee that every member of a team will follow the guidelines, as the compiler does not enforce them.

In summary, no previous technique exists for automatically verifying full conformance between a rich architectural specification and an implementation in a general-purpose programming language.

1.3   Consequences of Conformance Violations

The lack of automated conformance checking seriously compromises the benefits of architecture during implementation, testing, and software evolution. Communication integrity states that an architecture is complete. For example, the architecture should show all of the components that could possibly communicate with a given component. An engineer who is enhancing that component can effectively use this knowledge to make sure that the enhanced component interacts properly with all the existing components in the system. However, an engineer who cannot trust the architecture to be complete must fall back on more labor-intensive techniques for finding the other interacting components, or else risk introducing defects into the code.

Communication integrity is also crucial for preserving essential analysis properties. For example, in a security review, an architecture could be used to enumerate all of the possible information flows between secure and untrusted parts of the system. These information flows could then be analyzed in more detail in the source code. However, if communication integrity is not enforced, the architecture may not show all information flows that are present in the real system, and so the architectural analysis cannot be trusted to be correct. Without communication integrity, the source code of the entire system must be painstakingly analyzed, and the architecture provides little benefit at the implementation level.

In order to facilitate certain analyses or provide important implementation properties, an architect may select an *architecture style* for an application [GS93]. For example, the pipeline architectural style enables analysis of application throughput based on the performance of individual components, and the restrictions it entails on data flow can make reasoning about computation within a component easier. However, these benefits are lost if the implementation violates the constraints of the pipeline architectural style. Existing research efforts have shown how an architecture can be checked against style constraints [GAO94]; but these checks guarantee nothing about the properties of the implementation unless the implementation obeys communication integrity.

As the discussion above shows, *communication integrity is the fundamental conformance property relating architecture to implementation, upon which other conformance properties rely*. Communication integrity verifies *which* components communicate, providing the foundation for other architectural properties that show *how* these components communicate. For example, in a system with communication integrity, one can add a type system describing what kinds of data are exchanged, or a temporal protocol showing the order of messages between components, or pre- and post-conditions that place constraints on the data passed into and out of each operation. However, without communication integrity, it is difficult to specify or verify any of these architectural properties, because the specification will be incomplete if some communication paths are left out, and the verification will be unsound if some data and control flows are ignored. Thus, communication integrity is not merely one of many properties relating architecture to implementation; it is a core property essential for verifying conformance to many other architectural properties.

1.4   Design Goals

This dissertation presents the first technique for enforcing conformance to architectural structure in a general-purpose programming language. In order to make this statement more precise, it is necessary to define the goals of my system, ArchJava. This section describes several choices in the design space of enforcing conformance, and explains the choices made.

Flat vs. Hierarchical Architectures. Flat design notations, such as UML class diagrams [RJB98], are useful for understanding small-scale relationships between classes and objects in a system. However, they do not scale well to systems made up of hundreds of classes, because show only relationships between classes, not higher-level system design. ArchJava supports hierarchical architectures, so that a developer can describe the relationships between subsystems at the highest level, and then break these subsystems into parts recursively.

Class vs. Instance Architectures. Some tools show architectural relationships between fixed units of source code, such as classes [MNS01,MW99,SSW96,LR03]. Although class-based architectures can be helpful, understanding run-time relationships between objects is often crucial to understanding and evolving a program. In recognition of this, a number of recently developed architecture description languages model dynamic relationships between component instances [LV95,MK96,MOR+96,ADG98]. ArchJava models instance relationships between components so as to more precisely capture the dynamic structure of a program.

Visibility vs. Communication. A major line of work in programming languages has been developing expressive module systems that hide information within a component by restricting its visibility [MTH90,FF98]. Module systems such as that of ML have been proposed as architecture description languages [BHL94]. However, existing module systems only restrict the visibility of data, which is not sufficient to control all communication between components. For example, the definition of a function in component A may not be visible in component B, but if A passes the function to component B as a closure, then B can still invoke it. Thus, current module systems do not enforce communication integrity. The ArchJava system can be thought of as an extended module system that ensures that all inter-component communication is declared at the architectural level.

Dynamic vs. Static Checking. Systems such as Rapide provide a run-time monitor that can check conformance between an executing program and an architectural specification [LV95]. In contrast, the goal of ArchJava is to check architectural conformance statically (except for casts), so that when a program compiles correctly the developer has a guarantee that the program will conform to its architecture at run time.

Forbid vs. Allow Sharing. A number of architectural description languages discourage or forbid sharing data between components [LV95,MOR+96]. This implementation style may be appropriate for some applications, but many programs benefit from communication through shared data. ArchJava's design goal is to permit existing implementation techniques, including communication through shared data, but to require developers to declare whatever communication is present in their program.

Interface vs. Structure vs. Behavior. There is a spectrum of architectural specification and verification from interface checking through structural checking to behavioral checking. Existing module systems and type systems essentially check interface conformance between a module and its implementation. ArchJava carries this further to check that the connection structure at the architectural level is a conservative approximation of the communication structure in the run-time system. Other architecture description languages model some aspects of behavior as well, such as the temporal sequence of events in a component's interface [AG97]. However, previous ADLs have been unable to statically enforce either structural or behavioral properties in the implementation of a system.

Non-Technical Goals. ArchJava has non-technical goals besides the conformance goals listed above. ArchJava should be practical enough that I can evaluate it on existing programs. It should be expressive enough to support common programming idioms and techniques. Finally, it should provide value for software engineering tasks.

The Scope of ArchJava. There are a number of goals of previous architecture description languages that ArchJava does not attempt to address. ArchJava uses language-based techniques to address architectural conformance, so it does not explicitly support language interoperability. However, the techniques used by ArchJava could be applied to other languages, and architecture interchange languages such as xADL and Acme could be used to define architectures that include components written in ArchJava and components in other languages [GMW97,DHT02]. ArchJava currently models only architectural structure, leaving out important architectural issues like behavior modeling and architectural style that are addressed by other systems [GAO94,LV95,AG97]. It is my hope that the semantic connection that ArchJava enforces between architecture and implementation will eventually aid in specifying and verifying these properties in the implementation as well as the architecture.

Summary. Some previous systems statically enforce architectural conformance in a weaker setting—one that eliminates shared data, or models architecture in a limited way. However, ArchJava's goal of statically enforcing full conformance between an instanced-based architecture in a general-purpose programming language has been an important open problem that is solved in this dissertation.

1.5 Enforcing Conformance Using Types

This dissertation presents a new approach to enforcing architectural conformance: declaring the architectural structure of a system within a general-purpose programming language and using a type system to verify architectural conformance. This strategy creates a close link between architecture and implementation, sidestepping some of the problems of enforcing integrity between an implementation and a separate architecture description. I have chosen to extend the Java language, forcing us to confront the challenges posed by modern language constructs, and providing the opportunity to evaluate ArchJava on existing programs. In ArchJava, components are distinguished objects (not classes), allowing us to reason about architectural relationships between component instances.

In order to support static conformance checking, I developed a type system that ensures communication integrity. The type system is static in the same sense as that of Java: run-time checks are done only at casts and array writes. A type system can support more precise checking than an analysis-based technique, in part because the types express the programmer's intent, which is otherwise difficult to infer. Compared to global analysis-based techniques [LR03], ArchJava's type system uses only local checks, and is sound even

if not all code is available at compile time. Because the typechecker verifies simple, local rules, typechecking errors are understandable to the user.

In order to specify an architectural hierarchy, ArchJava builds on ideas from ownership type systems [NVP98]. Together with uniqueness types [Min96], ownership types also enable reasoning about how data is shared between components. Controlling sharing, in turn, allows ArchJava to constrain communication between components more strongly than the visibility constraints imposed by typical module systems.

Because ArchJava enforces architectural conformance, the benefits of architecture can be achieved not just at the architecture phase of a project, but throughout the software lifecycle. ArchJava's typechecking rules guarantee that the implementation will stay consistent with the architecture as the system evolves, so that engineers can be confident that they understand how components in the system interact when performing software evolution tasks. Architectural conformance in ArchJava provides a solid structural foundation for reasoning about the implementation of a system, and may provide benefits to automated analyses as well.

1.6   Contributions

The contributions of this dissertation are as follows:

- **Architecture within Implementation.** ArchJava is the first system to integrate a rich architectural description into a mainstream programming language. ArchJava extends Java with architecture description constructs, so that developers can specify an architecture during design and then fill in the architecture with Java implementation code. ArchJava includes components, ports, explicit connections, and other standard features of architecture description languages. Components are objects, supporting instance-based architectural reasoning, and ownership declarations are used to specify hierarchical relationships between components.

- **Lightweight Alias Control.** In order to control communication through shared data, I developed AliasJava, a lightweight and practical type system for controlling aliasing in object-oriented programs. AliasJava extends generalizes ownership type systems using ownership domains, which allow programmers to categorize objects into logical groups and then specify the permitted aliasing among those groups. AliasJava is both more precise and more flexible than previous ownership-based systems. For example, it is the first ownership system powerful enough to constrain aliasing between sibling objects that have the same owner. By default, AliasJava enforces the same owners-as-dominators property as other ownership type systems, but unlike other systems, it gives developers the flexibility to relax this property where necessary to implement common programming idioms such as iterators or events. AliasJava is also the first system to combine ownership-based encapsulation with uniqueness, a combination that is essential to expressing important architectural constraints as well as many practical idioms.

- **Enforcing Architectural Conformance.** ArchJava is the first system to enforce full structural conformance between a rich architectural description and general-purpose implementation code. Components may declare ownership domains in ports, allowing objects in these domains to be shared with connected components. ArchJava's type system builds on that of AliasJava, ensuring that components can only communicate via the interfaces declared in connections, or via objects in shared ownership domains that are declared in connected ports.

- **Formal Validation.** In order to validate the correctness of this approach, I present a core language ArchFJ incorporating the core constructs of ArchJava. Although ArchFJ is simpler than ArchJava, it is still quite expressive, modeling components, connections, ports, ownership, uniqueness, objects, mutable fields, and method calls. I formalize ArchFJ using a set of type judgments and rewriting rules, which collectively define the static and dynamic semantics of the language. I prove properties of type soundness and communication integrity for ArchFJ, increasing confidence that the full language enforces conformance as well.

- **Empirical Validation.** I have implemented a compiler that compiles ArchJava source down to ordinary Java bytecode, allowing programs to be run on any Java virtual machine. Using case studies on 4,000 lines of library and application code, I demonstrate that the alias-control system is practical enough to apply to existing code with few changes—the first empirical validation of any ownership-based alias control system. Two case studies on 10,000 line programs demonstrate the practicality and benefits of ArchJava's architecture constructs. This experience suggests that ArchJava typing rules encourage loose coupling between components, and may make evolution tasks easier by showing how components interact.

1.7   Thesis Statement and Outline

The thesis of this dissertation can be stated as follows:

> *A lightweight and practical type system can be used to enforce architectural conformance in general-purpose implementation languages, supporting more effective program reasoning and software evolution.*

The next chapter introduces ArchJava as an extension to Java, explaining the language through a series of examples and showing how the type system enforces communication integrity. Chapter 3 formalizes the core of the language using type judgments and rewriting rules, and gives a formal proof that the type system enforces architectural conformance. Chapter 4 evaluates the practicality and the utility of ArchJava through several case studies. Chapter 5 compares ArchJava to related work, and Chapter 6 critiques the ArchJava project: what was effective about the language, and what lessons have been learned for future projects. The last chapter concludes with a discussion of future work.

Chapter 2

The ArchJava Language

This chapter presents the ArchJava language and explains informally how the type system enforces communication integrity. The first section of the chapter provides a brief tour of ArchJava through an example architecture, providing a broad overview of the language before other sections dive into details.

Section 2.2 begins by presenting AliasJava, a stand-alone subset of ArchJava that allows developers to specify and verify aliasing relationships in an object-oriented system. Controlling aliasing is at the core of enforcing communication integrity, because unrestricted object references permit non-local communication between components. AliasJava uses type annotations to specify references that are unique or unaliased, or references that are confined to within a particular ownership domain. These constructs allow programmers to specify how objects are shared between architectural components.

Section 2.3 builds on AliasJava by presenting language features for specifying architectural structure. Developers use component classes in place of regular classes to indicate that the objects created from these classes are part of the architecture of the system. Components communicate through ports, which are endpoints for communication that declare detailed interfaces. Connections link the ports of components together, allowing the components to communicate. Components, ports, and connections are all standard constructs of architectural description languages; ArchJava is unique in integrating them into a mainstream programming language, so that a type system can be applied to verify communication integrity.

Section 2.4 defines communication integrity more precisely and shows how the architecture constructs combine with the aliasing annotations to enforce integrity. ArchJava has been implemented, and the techniques used are discussed briefly in Section 2.5. Finally, section 2.6 discusses the differences between the current versions of ArchJava and AliasJava and previously published descriptions of the languages [ACN02a,ACN02b,AKC02].

2.1   A Brief Tour of ArchJava

ArchJava models architectures using *components* that communicate through *connected ports* and objects in shared *ownership domains*. For example, the rectangle at the top of Figure 1 shows the architecture of a hypothetical word processor. At the highest level, the word processor is made up of two component objects, shown as nested rectangles: the user interface and the document. Just as objects are instantiated from classes in Java, component objects are instantiated from *component classes* in ArchJava. ArchJava's component classes can have all the features of ordinary Java classes, but in addition can contain architectural modeling features. These architectural modeling features allow developers to specify the communication between components more precisely than is possible with ordinary classes. The code
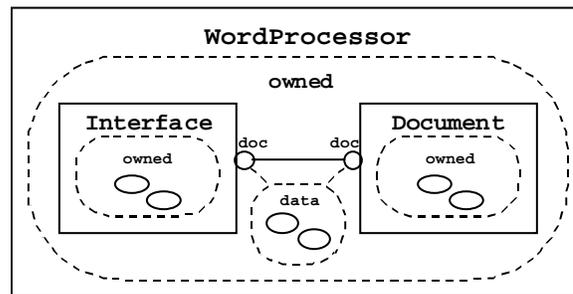
```
public component class WordProcessor {        public component class Document {
  domain owned;                                  port doc {
  private owned Document doc;                       domain data;
  private owned Interface ui;                       provides unique Position search(
  connect doc.doc, ui.doc;                                                lent String q);
  // additional source code...                      provides void insert(lent Position loc,
}                                                                        data Text text);
                                                  }
public component class Interface {
  port doc {                                       domain owned;
    domain data;                                   owned DocumentIndex<data> index;
    requires unique Position search(
                         lent String q);           // additional source code...
    requires void insert(lent Position loc,      }
                         data Text text);
  }
  // additional source code...
}
```

Figure 1. The `WordProcessor` component class describes an architecture made up of an `Interface` and a `Document` component that are part of its `owned` ownership domain. The two subcomponents of the word processor communicate through connected `doc` ports. The `doc` port of the `Interface` component class declares two methods that it requires the `Document` to implement, while the corresponding port in `Document` declares that it implements these methods. Both ports declare the ownership domain `data`, representing document information shared between the two objects. All fields, parameters, and results of reference type are annotated with aliasing information—either the ownership domain that owns the object in the field, or `lent` (indicating a temporary alias to an object), or `unique` (representing the sole reference to an object).

below Figure 1 shows three component classes: the WordProcessor[1] is a template for the entire application, while the Interface and Document component classes are templates for the user interface and document components.

In the diagram in Figure 1, the user interface and document component objects are nested within the word processor, indicating they are parts of the word processor. In order to declare this nesting relationship within the source code, the word processor declares an ownership domain **owned** that holds the objects nested within it. In the diagram, ownership domains are represented with rounded, dashed rectangles. In the source code, ownership domains are declared with the keyword **domain**. The document and user

---

[1] In the text and examples, program code is shown in a constant-width font, and language keywords are in **bold**.

interface also declare ownership domains (also named **owned**) to store their nested components and objects.

In order to indicate that the document and user interface component objects are in the **owned** domain of the word processor, the type of the `doc` and `ui` fields are annotated with the **owned** domain. This annotation indicates that the objects to which these fields refer are part of that domain. Section 2.2 describes ownership domains in more detail, and explains how they can be used to restrict aliasing.

ArchJava uses ports to declare the interfaces used for communication with external components. Port interfaces are more precise than ordinary Java interfaces in three ways. First, ports declare not just the methods that a component provides (for example, the `search` and `insert` methods in the `doc` port of `Document`) but also the methods that component requires other components to implement (for example, the `Interface` component requires an implementation of those same methods).

Second, ports can declare ownership domains that contain objects that are shared between the component and the outside world. For example, the document and user interface each declare the `data` ownership domain in their `doc` ports, representing document data that is shared between the components. The arguments and result types of methods declared in ports can be annotated with these ownership domains. For example, the `Text` argument of the `insert` method must be part of the `data` ownership domain. Figure 1 shows other aliasing annotations as well. The `search` method returns a `Position` that is **unique**, indicating that there can be no aliases to the position object. The `insert` method accepts a `Position` parameter that is **lent**, meaning that it may be used for the duration of the method call but persistent aliases to it may not be created. These annotations are discussed in more detail in the next section.

Finally, a component can have multiple ports, distinguishing the protocols used to communicate with multiple external components. A Java class can partly achieve this effect by implementing multiple interfaces, but there is no way to tell which clients use which interfaces. ArchJava makes connectivity information explicit by declaring connections between components. For example, the word processor declares a connection linking the `doc` ports of its subcomponents. This connection binds the required methods in the user interface to the corresponding provided methods in the document, so that these components can communicate.

The architecture constructs of ArchJava allow developers to model hierarchical architectures, and precisely specify the control flow and data sharing between them. Section 2.3 describes these architectural constructs in more detail, and explains how they can be used to enforce communication integrity.

2.2 AliasJava

The AliasJava type annotation system is designed to support reasoning about aliasing in large object-oriented software systems. AliasJava allows developers to express the lack of aliasing through *uniqueness*, and controlled aliasing through *ownership*. AliasJava expresses aliasing constraints using annotations on reference types. Since values of primitive type cannot be aliased, these types are not annotated. The full ArchJava language, discussed in Section 2.3, builds on this foundation by adding an architectural framework.

Each object in the system is either *unique*, or else part of an *ownership domain* that is declared by another object. A unique object has only one persistent reference to it, although temporary *lent* aliases may be created. Objects within a single ownership domain can refer to one another, but references can only cross domain boundaries if the programmer specifies a *link* between the two domains when they are created.

Each object declares a set of named ownership domains. These domains are nested within the domain that owns the object, so that ownership defines a forest of trees where each parent owns its children and the roots of the tree are unique. Unique objects may be assigned to an ownership domain, attaching one ownership tree as a subtree of another. The subsections below explain how these relationships are declared as type annotations within the program source and how they are enforced by the type system.

In this section, I present the annotations as a type system for Java programs that provides global guarantees about aliasing. However, adding alias annotations to a large legacy program may require significant effort. The annotations can also be applied to verify local properties within a subsystem, treating the annotations at the edge of the subsystem as unchecked assertions. I use this methodology in my case studies in Section 4. A promising alternative is inferring alias annotations for a closed subset of the program automatically, and other work presents early results in this direction [AKC02].

Subsection 2.2.1 describes the alias annotations through a series of examples. A more precise description of the core annotation system is provided by the formal semantics in section 3. Subsections 2.2.2 and 2.2.3 describe the properties guaranteed by AliasJava's annotation system, and how the annotations integrate into the full Java language. Finally, subsection 2.2.4 shows more examples of the annotation system in order to illustrate its expressiveness.

```
class LinkedList {                            unique LinkedList list =
  private unique Object item;                      new LinkedList(new Object(), null);
  private unique LinkedList next;             list=new LinkedList(new Object(), list);
                                              unique Object o = list.getItem();
  public LinkedList(unique Object o,          list = list.getNext();
                    unique LinkedList n) {
    item = o; next = n;
  }
  public unique Object getItem() {
    unique Object temp = item;
    item = null;
    return temp;
  }
  public unique LinkedList getNext() {
    unique LinkedList tempNext = next;
    next = null;
    return tempNext;
  }
}
```

Figure 2.   A linked list class with `unique` links and items

2.2.1   Annotations for Data Sharing

Unique. When an object is first created, it is *unique*—that is, there is only one reference to the object. The type annotation **unique** describes a reference that does not have persistent aliases. Figure 2 illustrates uniqueness through a linked list class where all of the elements and all of the links are **unique**.

In general, after a **unique** variable or field is read, the source location must be dead (that is, unused by subsequent code)—otherwise the read reference would be an alias of the supposedly unique source. A standard intraprocedural live variable analysis is used to verify this criterion for **unique** local variables. When a **unique** field is read by a method, that method must set the field to another value before executing any statement that could result in reading the original value of the field a second time, such as a method call or exception-throwing expression. For example, in Figure 2, the getItem method sets the item field to **null** so that no aliases to the **unique** value exist when the value is returned.

In AliasJava, **unique** can be considered a universal source: **unique** values can be assigned to a location with any other data sharing annotation. The converse is not true, as the other data sharing annotations do not guarantee that a value is unique.

Ownership Domains. An *ownership domain* is a group of related objects that are conceptually part of the object that declared the domain. Each object is part of a single ownership domain. An object's *owner* is the object that declared its ownership domain, and its ownership domain is its *owning domain*.

Ownership domains are illustrated using a simple address lookup application, such as one might find on a PDA. The diagram at the top of Figure 3 illustrates the ownership domains in the application at run time.

```
public class AddressApplication {            public class Table<key,value>
  domain name, address, owned;                      extends Map<key,value>
  link owned->name, owned->address;                 assume owner->key, owner->value {
                                               domain owned;
  private owned Table<name,address> addrBook;  link owned->key, owned->value;

  private address Addr lookup(name String n) { private owned Set<key> keys;
    return addrBook.get(n);                    private owned List<value> values;
  }
                                               public value Object get(key Object k) {
  // application UI, etc., not shown...           for (int i = 0; i < keys.size(); ++i)
}                                                   if (keys.get(i).equals(k))
                                                      return values.get(i);
                                                 return null;
                                               }
                                             }
```

Figure 3. The `AddressApplication` class declares three ownership domains: `name`, `address`, and `owned`. The `owned` domain is linked to the `name` and `address` domains, allowing the `owned` `addressBook` object to refer to names and addresses. The `addressBook`'s formal domain parameters `key` and `value` are bound to the `name` and `address` domains, respectively.

Solid circles represent objects, and dashed circles represent ownership domains. The AddressApplication object declares three ownership domains: the name domain holds names that the user may want to look up, the address domain holds address objects, and the **owned** domain holds the application's addrBook object. The addrBook object is a table mapping names to addresses, and declares a single ownership domain **owned**, which holds its internal data structures.

The domains in AddressApplication are declared in the second line of Figure 3 using the **domain** keyword. In order to indicate that an object is part of a domain, the type of the object is annotated with the domain name. For example, in Figure 3, the addrBook is an object of type Table that is part of the **owned** domain. When an object is instantiated, it is given fresh ownership domains that are distinct from the ownership domains of all other objects. For example, the **owned** domain inside one AddressApplication object is distinct from the **owned** domain inside other AddressApplication objects, and is also distinct from the **owned** domain of Table objects such as addrBook. In AliasJava, the **owned** domain is a default domain that is built into each object, and need not be explicitly declared, although Figure 3 does so for clarity.

Domain Parameterization. The `Table` object needs a way to refer to the ownership domains that hold the table's keys and values. These domains cannot be declared inside the `Table`, because the keys and values are typically owned by the table's client. Instead, the `Table` is given formal domain parameters for the `key` and `value` domains, and the implementation of the table uses these names to refer to the domains of key and value objects. Clients of the table must supply domains as actual parameters when they refer to a `Table`. For example, the declaration of `addressBook` instantiates `key` with `name` and `value` with `address`.

Link Declarations. Sometimes objects in one ownership domain need to refer to objects in another ownership domain. For example, the `addrBook` in the **owned** domain needs to refer to name and address objects in the `name` and `address` domains. A *link declaration* allows references from one domain to another. Each class must declare all allowed links between a domain it declares and any other domains in scope. The diagram in Figure 3 shows link declarations as arrows. For example, the arrows from the **owned** domain of `AddressApplication` to the `name` and `address` domains allow references in the direction of the arrow (but not the reverse direction). The third line of code in Figure 3 shows how these links are declared in source code using the `->` operator. As a shorthand, bi-directional links can be declared with the `-` operator.

If a class relies on a link between two formal domain parameters, it must state this explicitly using an **assume** clause in the class declaration. For example, `Table` objects need to store references to their keys and values, and so the owner of the table must be linked to the `key` and `value` domains. Thus, the table's class declaration states the assumption that the table's owning domain (denoted with the **owner** keyword) is linked to `key` and `value`.

Defaults. Explicitly specifying linking assumptions and link declarations can be inconvenient, so the AliasJava language includes defaults that cover the most common declarations. Since the **owned** domain is declared implicitly inside every object, objects that don't need to distinguish different groups of objects don't have to declare any domains at all. The default annotation for object fields is **owned**, so if this domain is appropriate, no annotation is necessary.

Since an object typically needs to refer to objects owned by its formal domain parameters, all classes assume that the **owner** domain is linked to every domain parameter. To make the **owned** domain convenient, AliasJava also assumes that the **owned** domain is linked to every domain parameter of the class, as well as to every domain declared within the class.

16

```
class Singleton {
  private static shared Singleton val
                      = new Singleton();

  public static shared Singleton get() {
    return val;
  }
  public void doSomething() {
    // application specific code
  }
}

shared Singleton s = Singleton.get();
s.doSomething();
```

Figure 4. A shared Singleton object

```
boolean contains(lent LinkedList head,
                          int i) {
  for (lent LinkedList list = head;
       list != null; list = list.next) {
    lent Integer item = (Integer) list.item;
    if (item.intValue() == i)
      return true;
  }
  return false;
}
```

Figure 5. A method that uses a lent reference to traverse a linked list looking for an integer

With these defaults, none of the **assume** and **link** declarations in Figure 3 are necessary, and none of the **owned** domains need be declared.

Method Parameterization. Sometimes it makes more sense to parameterize a single method rather than the whole class. For example, the Table class might include the method below, which returns an iterator over the keys in the table. The caller of getKeys specifies the iter domain, which will own the iterator. As with class parameters, the method must declare the linking assumptions it makes between the method's domain parameter and other domains in scope. The method can also declare links between the domain parameter and domains declared in the class. For example, since the iterator must refer both to the keys and the internal set that stores the keys, the getKeys method assumes that the iter domain is linked to the key and **owner** domains. It then declares a link between the iter domain and the internal **owned** domain, allowing the iterator to access the set of keys.

```
public iter Iterator<key> getKeys<iter>()
               assume iter->key, iter->owner {
  link iter->owned;
  return new TableIter<key,owned>(keys);
}
```

Shared. Figure 4 illustrates the Singleton design pattern, used to create a single instance of an object that is used throughout an application [GHJ+94]. Singleton objects are sometimes shared throughout a program, and thus cannot be confined by an owning object. References to such objects are annotated **shared**, representing the fact that these objects may be shared globally. Formally, **shared** is modeled as an ownership domain that is an implicit parameter of all objects. Unfortunately, little reasoning can be done about **shared** references, except that they may not alias non-shared references. However, shared references are essential for interoperating with existing run-time libraries, legacy code, and static fields, all of which may refer to aliases that are not confined to the scope of any object instance.

Lent. Figure 5 shows a method that could be part of the `LinkedList` class from Figure 2. This method checks if an integer is stored in a linked list that is made up of **unique** `LinkedList` and `Integer` objects. This would be difficult to express with the annotations presented so far, because `contains` would have to destroy the linked list while traversing it in order to avoid creating aliases to the links and elements in the list. Instead, the method uses the **lent** annotation to create temporary aliases to the unique objects in the list. These aliases must be destroyed when the `contains` method returns, so that the uniqueness of the linked list is preserved across calls to `contains`.

As shown in this example, a **unique** object can be passed to any method as a **lent** parameter. The called method can pass on the object as a **lent** parameter to other methods, but cannot return it or store it in any field. Thus, the uniqueness of the **lent** object is restored when the method returns. The **lent** type can also be used to temporarily pass an **owned** object to an external method for the duration of a method call, without any risk that the outside component might keep a reference to that object. Therefore, **lent** can be considered a universal sink: values with any alias type annotation may be assigned to a **lent** location. The converse is prohibited: **lent** values may only be assigned to other **lent** locations. Lent can be thought of a restricted capability that can be used to access an object, but cannot be used to store the object in a field. Lent is the default annotation for method arguments and local variables, and may be omitted.

Other annotations. In designing the annotation system, I chose to focus on precisely specifying the aliasing relationships between objects in the system. Using this criterion, I decided not to include a few annotations that are used in some of the related work. Although package-based confinement [BV99] provides a middle ground between the global **shared** domain and domains that are local to an object, I chose not to include it because object ownership is a stronger property and I wanted to keep the system simple. Read-only annotations [NVP98,MP99,BNR01,BR01] can also express useful invariants about a system, but they are orthogonal to aliasing and so were not included in the design. These annotations could probably be added to the system in a natural way.
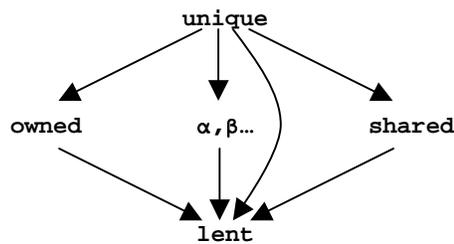
Figure 6.   Value flow between alias annotations

Summary.   Figure 6 shows the constraints that the type annotations place on value flow.   An arrow indicates that data can flow between variables with two annotations in the direction shown.   The figure shows clearly that **unique** is a universal source (any variable can be assigned a **unique** value), and that **lent** is a universal sink (**lent** variables can be assigned a value with any type annotation).   The other type annotations (the built-in domains **owned** and **shared**, as well as declared domains α, β, etc.) must be kept separate from each other.

2.2.2   Properties

AliasJava ensures uniqueness and ownership invariants that restrict the aliasing patterns that can occur during program execution.   Section 3 proves these invariants for a subset of the full language.   The uniqueness property states that variables and fields with the **unique** annotation hold unique references (ignoring temporary **lent** aliases).   In the presence of concurrency, enforcing the uniqueness property requires synchronization on unique field reads, as discussed in section 2.2.3.

> **Uniqueness.**   At a particular point in dynamic program execution, if a variable or field
> that refers to an object o is annotated **unique**, then no other field in the program refers
> to o, and all other local variables that refer to o are annotated **lent**.

ArchJava enforces two key properties that restrict aliasing between ownership domains.   The first property is *link soundness*:

> **Link Soundness.**   If D_1 and D_2 are ownership domains, and an object that is owned by D_1
> refers to an object that is owned by D_2, then there must be an explicit declaration linking
> D_1 to D_2.   Furthermore, if D_1 and D_2 are ownership domains that are visible in the same
> scope, and an object that is *transitively* owned by D_1 refers to an object that is *transitively*
> owned by D_2, then there must be an explicit declaration linking D_1 to D_2.

Link soundness ensures that inter-domain references are only present between linked domains.   Link soundness is enforced by making sure references between objects are legal given the linking assumptions and declarations in scope, and checking that the linking assumptions of a class are fulfilled whenever that

```
public class Class {
  private shared Object signers owned [];

  // compiler error: cannot return owned state to clients
  public owned Object[] getSigners() {
    return signers;
  }
}
```

Figure 7.   In an early version of the JDK, the `getSigners` method of `Class` returned the internal array holding the signers of a class, allowing clients to modify the list of signers. If the array had been declared `owned` using AliasJava, the type system would have caught this security hole.

class is instantiated. For example, in Figure 3 it is legal to instantiate `Table` with actual parameters `name` and `address` and an owning domain **owned**, because the owned domain is linked to `name` and `address`, fulfilling the assumption stated in the class header of `Table`. Chapter 3 gives the formal typechecking rules that enforce link soundness.

As the second part of the definition shows, link soundness constrains not just references from a domain D_1 to another domain D_2, but all references from the ownership tree rooted at D_1 to the ownership tree rooted at D_2. In order to enforce this hierarchical relationship, a class that links one of its ownership domains to one of its ownership parameters must assume a corresponding relationship between its owner domain and that ownership parameter. For example, in Figure 3 the link between `owned` and `key` in `Table` is only legal because the table assumes a link between its owner and the `key` domain.

Ownership domains provide a stronger kind of encapsulation than Java's **private** and **protected** modifiers, which restrict only the visibility of a field, not the accessibility of the object inside the field. For example, Figure 7 illustrates a defect in an early version of the Java Standard Library that was caused by a **private** object escaping to clients. In this bug, the security system function `Class.getSigners()` returned a pointer to the internal array used to store the principals that had signed a class, rather than a copy. Clients could then modify the array, making the class appear to have been signed by a trusted principal, and thus potentially allowing malicious applets to pose as trusted code. In Figure 4, the array has a type indicating an **owned** array of **shared** signer objects (the array notation is discussed further in subsection 2.2.3). With this declaration, the compiler would have caught this bug at compile time, because public methods cannot return objects owned by internal ownership domains. The bug is fixed by returning a copy of the `signers` array from `getSigners`, rather than the actual array.

The encapsulation property enforced by AliasJava is called *capability-based encapsulation* [AKC02]:

> **Capability-based Encapsulation.** Object O_1 cannot refer to object O_2 unless a name (or *capability*) for O_2's ownership domain is in scope in object O_1.

An ownership domain name acts as a capability for accessing objects that are part of the domain. If an object can't name a domain, it can't refer to objects in that domain. Initially, each object has the unique capability to access the objects in the ownership domains that it declares. The object can share a capability by passing the corresponding domain as an actual ownership parameter of another object. If an ownership domain is never passed as an ownership parameter, then only the owner can refer to objects in that domain. Conversely, an object that has no formal domain parameters can only refer to the objects that it owns.

To ensure that capability-based encapsulation is meaningful, ownership annotations must be consistent over time:

> **Ownership Soundness.** At a particular point in dynamic program execution, if a variable or field referring to object `o` has an ownership annotation denoting ownership domain `d`, then all other variables or fields that refer to `o` at any subsequent point in dynamic program execution, are either annotated **lent** or have an ownership annotation denoting the same domain `d`.

Link soundness and capability-based encapsulation together place stronger constraints on aliasing than either could alone. Even if there is a link from one domain to another, objects in the first domain must have a capability for the second domain in order to refer to objects in that domain. Link soundness ensures that a domain cannot be shared arbitrarily, but can only be passed as a domain parameter to objects whose owner is linked to the domain.

Program Reasoning. Through uniqueness and ownership domains, AliasJava supports static, source-level human and automated reasoning about aliasing in object-oriented systems. This dissertation applies AliasJava's alias-control system to verify the communication integrity property, ensuring that the implementation of a system is consistent with its architectural design.

Other researchers have applied ownership-based systems to a variety of other problems, including eliminating data races [BR01] and deadlocks [BLR02], supporting code updates [BLS02], checking effects [CD02], proving representation independence [BN02], and verifying program invariants [SD03]. Most of these systems rely on the *owners as dominators* encapsulation property in order to address these problems:

> **Owners as Dominators.** If object `o` refers to another object `o'`, then `o` is inside (i.e., transitively owned by) the owner of `o'`.

The owners as dominators property ensures that there are no pointers from the outside of an object to the inside—only the other direction. The owners as dominators property prohibits many useful idioms, including iterators and event callbacks, but it is nevertheless useful for proving certain properties.

AliasJava occupies a unique point in the design space: with the appropriate link specifications, it can be used to enforce owners as dominators, but with other link specifications, it is flexible enough to support idioms like iterators and event callbacks. For example, to support owners as dominators, a class should specify a link from **owned** domain to each of its ownership parameters, but never a link from an ownership parameter to the **owned** domain (or any other internal domain). In fact, this set of links is the default in the AliasJava system; a program with no explicit linking specifications enforces owners-as-dominators. Linking assumptions between ownership parameters in AliasJava are analogs to the assumptions in other systems stating that one parameter is within another. Thus, AliasJava can be used to enforce the same properties as other ownership-based systems in the default case, while still providing additional flexibility if it is needed.

2.2.3   Java Integration

The Java language has several features that present challenges for an alias control system. I discuss how AliasJava handles of a number of these features below.

Subtyping. Java's declared subtyping relation is extended with alias annotations. When a class is defined, it must provide values for the ownership parameters of the classes and interfaces it extends and implements; these values can be any of the ownership parameters of the subclass, or **shared**. For example, a class declaration might look like: **class** C<α,β,γ> **extends** B<α,β> **implements** I<γ>. When a method or field is overridden, the overriding member must declare its parameter types and return type with annotations that exactly match the overridden member, under the ownership parameter mapping induced by the inheritance declarations. In general, it would be sound to override a method with covariant parameter types and a contravariant result type, but AliasJava requires an exact match to be consistent with Java's existing semantics for overriding methods.

This. Since the current object **this** is an implicit argument to all instance methods, its type annotation must be specified. This is done with an annotation that comes immediately after the argument list. This type may be one of **shared**, **unique**, **lent**, or an ownership parameter. Use of **this** within the method must be consistent with its annotation, and at method calls, the receiver is treated as another parameter that must follow the rules for the **this** alias annotation. Because the vast majority of methods have a **lent** annotation for **this**, **lent** is the default in the system and need not be explicitly specified.

Constructors. Like methods, constructors must specify an alias annotation for **this**. Semantically, a new statement is treated as an allocation of a **unique** object followed by a method call to the constructor for initialization. For example, the expression **new** `Foo()` is modeled as the sequence of statements **unique** `Foo temp =` **allocate** `Foo; temp.Foo().` If the constructor's **this** annotation is **lent**, the allocated object will remain **unique**. If the constructor's **this** annotation is **shared**, the allocated object will be **shared**, and if the constructor's **this** annotation is **unique**, the result of the **new** expression must be dead. Thus, the alias annotation of a newly allocated and constructed object will be **unique** in the common case where the constructor's **this** annotation is **lent**.

Inner Classes. Non-static inner classes implicitly import the parameters of their surrounding class. The inner class can have its own additional parameters, if necessary. Thus, the fully qualified type of an inner class is of the form `package.EnclosingClass<α…>.InnerClass<β…>`. An inner class can refer to the domains declared in the enclosing class (including **owned**) using a name of the form `EnclosingClassName.domainname`. The domains of the inner class can be referenced without qualification. Anonymous classes defined within a function may not access **unique** or **lent** local variables from the function's scope, because such accesses could create internal persistent references stored in the inner class object, which may violate the type system's invariants.

These special rules do not apply to **static** classes defined within another class. Such classes do not have an implicit pointer to an object of the enclosing class, and so they follow the same rules as ordinary classes, with no special access to their enclosing class.

Static Fields. Static fields are not associated with any particular object instance, and so they cannot be declared with a domain as the type annotation (also recall that no field may have a **lent** annotation). Static fields can be **unique** if they are read and written in a way consistent with the **unique** annotation.

Concurrency. Concurrency is orthogonal to this work, except for possible data races when reading a unique field. In the presence of concurrency, access to unique fields must be synchronized to prevent two threads from reading a unique variable simultaneously, creating two aliases of a supposedly unique value. In order to guarantee uniqueness in the presence of concurrency, the compiler will eventually include a *concurrent mode* that performs additional checks. These checks ensure that a **unique** value can flow from an object field into another non-**lent** location only within a block of code synchronized on the object

whose field is being dereferenced (or the field's declaring class, in the case of static fields). The field that was read must also be set to another value before the end of the synchronization block.

Casts. Because a class may extend a class that has fewer parameters, ownership parameters may be hidden when an object is treated as its supertype. For example, a `List` object with a single parameter `data` may be upcast to type `Object`, which has no ownership parameters. Later, the programmer may want to downcast a variable `o` of type `Object` to type `List` with the expression `(List<data>)o`. In order to preserve soundness, the run-time system must check both that object `o` is of type `List`, and also that the `List`'s ownership parameter is `data`.

In the implementation, a tag object is allocated for each declared domain. Each parameterized object stores the tag objects representing the actual domains for each of its ownership parameters. Run-time domain tags are also passed to methods that have ownership parameters. This run-time information is assigned at object-creation time. Note that AliasJava does not need to store the domain of each object in the system; the system incurs space overhead only for objects that are parameterized.

When an object is cast to a parameterized type, the run-time owner for each of its parameters is checked against the corresponding owner specified in the cast, and an `AliasCastException` is thrown if the check does not succeed. In this way, AliasJava supports upcasts and downcasts in a way that does not violate the semantics of the type annotations.

Arrays. An array must be given an alias type for each array dimension. The alias type of the objects inside the array is given next to the type of the objects, while the modifier for each dimension of the array is next to the corresponding brackets. For example, the variable declaration α `Stack<`β`>` **owned** `[ ]` **unique** `[ ]` `array` refers to a **unique** array of **owned** arrays of α stacks that hold objects of alias type β. This syntax is consistent with the syntax for **const** arrays in C++. An array dereference of the form `array[0]` would yield a value of type α `Stack<`β`>` **owned** `[ ]`.

Following Java, AliasJava supports covariant subtyping for arrays. In order to preserve type soundness, it must do a run-time alias check whenever an object is stored into an array, to ensure that the dynamic ownership parameters of the object are compatible with the dynamic ownership parameters of the array. This check uses the same run-time alias annotation information that is used to support sound casts, as discussed above. AliasJava keeps track of the relationship between arrays and their ownership parameters using a global hash table, since this information cannot be stored in the array itself. The keys in the hash

24

```
interface Iterator<element> {
  element Object next();
}

public class List<element> {
  private owned Link<element, owned> front;
  void add(element Object e) { ... }
  public i Iterator<element> iterator<i>()
             assume i->element, i->owner
             link i->owned {
    return new ListIter<element,owned>
                               (front);
  }
}

class ListIter<element, link>
      implements Iterator<element>
      assume link->element {
  private link Link<element, link> cur;
  public element Object next() {
    element Object e = cur.o;
    cur = cur.next;
    return e;
  }
}
```

```
public class Lexer {
  owned InputStream stream;
  Lexer(unique InputStream s) {
    stream = s;
  }
  unique Token getToken() { ... }
}

void lexerClient() {
  unique InputStream stream =
                new FileInputStream(file);
  unique Lexer l = new Lexer(stream);
  l.getToken();
}
```

Figure 8. A `List` class and an iterator over the list

Figure 9. A `Lexer` class that uses an `InputStream` as part of its representation. The `InputStream` is passed to the constructor as a `unique` reference.

table are weak references, so that the arrays (and their ownership information) can be reclaimed by the garbage collector when they are otherwise unreachable.

The Java Standard Library. AliasJava is implemented on top of the standard Java Virtual Machine (JVM), and applications can use the Java standard library that is provided with the VM. Unfortunately, this means that Java's reflection interfaces provide a way to get around the alias type system. This could be remedied by replacing the existing reflection library with one that dynamically checks for violations of the alias type system.

Another issue is that since I did not modify the standard library, the runtime system does not record run-time ownership parameter information for parameterized classes and methods created and called by the standard library code. Thus, the parameter information for some methods and objects will be missing at some run-time casts. In the implementation, these casts always succeed, but a number of other choices are possible in principle. In the future, I hope to apply an improved version of alias annotation inference [AKC02] to the standard library, and provide the annotated library along with the ArchJava distribution.

2.2.4 Examples

In this subsection, I present three examples that demonstrate the expressiveness and benefits of the annotation system.

Iterators. Iterators are a challenge to many ownership-based alias control systems. For example, none of the early ownership type systems supported iterators [NVP98,CPN98,CNP01], and more recent systems support them only as inner classes [Cla01,BLS03] or as objects that cannot escape the stack [CD02]. Figure 8 shows how a `List` class can be defined to return an `Iterator` object that can access its internal representation (the links in the list) without exposing that representation to clients. When the `List` class creates a `ListIter`, it instantiates the second ownership parameter of `ListIter` with **owned**, thereby delegating a capability to access the list's representation. The `ListIter` is then returned as an object of type `Iterator`, which hides access to the links in the list. Clients of the `Iterator` cannot access these links through the `Iterator` interface, nor can they cast the `Iterator` to `ListIter`, because the `List` has not given them a capability to access its representation. Furthermore, the programmer is protected from accidentally returning a `ListIter`, because the `ListIter`'s argument **owned** is an internal domain of `List` and therefore may not appear as part of any type in the public interface of `List`.

Uniqueness and Ownership. The combination of the **unique** annotation with ownership annotations is crucial to the expressiveness of the annotation system; it allows us to express important idioms that neither class of annotation system could alone. For example, the `Lexer` class in Figure 9 accepts an input stream that becomes part of its representation. The implementation of the `Lexer` relies on the state of the `InputStream`, and therefore the specification of `Lexer` should require that external clients do not modify the state of the stream after passing it to the lexer.

In AliasJava, the `InputStream` argument to `Lexer`'s constructor is **unique**, forcing the client to give up its other non-**lent** references to the stream. The `InputStream` is then captured into the lexer as an **owned** reference, which is encapsulated within the lexer object.

2.2.5   Summary

AliasJava's annotations allow programmers to express and enforce important aliasing properties such as uniqueness, encapsulation, temporary sharing, and persistent sharing. The next section describes how programmers can express the architectural structure of a system, and shows how these aliasing annotations can be used within that framework to describe important classes of architectural communication.

2.3   Architecture Constructs

The ownership domains of AliasJava are sufficient to define hierarchical architectures, and to state a communication integrity theorem that all communication between ownership domains follows linking specifications. Thus, ownership domains provide the basis for the key technical result of this dissertation. However, although this model of architecture specifies communication structure, it provides little insight into the way that objects in different domains communicate.

In order to specify architectural communication more precisely, software architecture researchers have developed *component*, *port*, and *connection* abstractions. Architectural elements are modeled by components, distinguished objects that communicate in a more structured way. Ports represent the endpoints of communication between components; they show method calls that are sent and received by a component, and declare ownership domains that are shared between those components. Explicit connections link ports together, showing which components communicate and using what protocols.

ArchJava adds new language constructs for components, ports, and connections in order to allow developers to specify architecture in a precise way. The rest of this section describes the language design, describing by example how to use these constructs to express software architectures. Throughout the discussion, I 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.

2.3.1   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 10.

A *port* represents a logical communication channel between a component and one or more components to which it is connected. Ports declare two sets of methods, specified using the **provides** and **requires** keywords. A *provided* method is implemented by the component and is available to be called by other components connected to this port. Conversely, each *required* method is provided by some other component connected to this port. A component can invoke one of its ports' required methods by sending a message to the port. For example, the `parse` method calls `nextToken` on the parser's `in` port.

```
public component class Parser {
  public port in {
    domain symbol;
    provides void setInfo(symbol Token t, symbol SymInfo s);
    requires symbol Token nextToken() throws ScanException;
  }
  public port out {
    domain symbol;
    provides symbol SymInfo getInfo(lent Token t);
    requires void compile(unique AST<symbol> ast);
  }

  public void parse() {
    symbol Token tok = in.nextToken();
    unique AST<symbol> ast = parseFile(tok);
    out.compile(ast);
  }

  unique AST<symbol> parseFile(symbol Token lookahead) { ... }
  void setInfo(symbol Token t, symbol SymInfo s) { ... }
  symbol SymInfo getInfo(lent Token t) { ... }
  ...
}
```

Figure 10. 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.

Because ArchJava is built on top of the AliasJava annotation system, the parser code is annotated to describe aliasing constraints on the data structures it manipulates. Unlike regular classes in ArchJava, component classes do not have ownership parameters. Instead, they specify data sharing more explicitly by declaring ownership domains in their ports, which represent groups of objects shared with other components. For example, in Figure 10, both the `in` and the `out` ports of `Parser` declare the domain `symbol`, representing tokens and symbol table information in the compiler that is shared with other components. Domain declarations of the same name in the same component class, such as the declarations of `symbol` in the `in` and `out` ports, refer to the same domain.

ArchJava's ports specify both the services implemented by a component and the services a component needs to do its job. Required interfaces make dependencies explicit, reducing coupling between components and promoting understanding of components in isolation. Ports also make it easier to reason about a component's communication patterns.
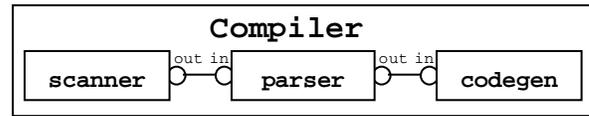
```
public component class Compiler {
  private final owned Scanner scanner = ...;
  private final owned Parser parser = ...;
  private final owned CodeGen codegen = ...;

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

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

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

Figure 11. 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.

2.3.2   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 owned by another component. Singleton subcomponents can be declared as **final** fields of component type.

Figure 11 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 software evolution tasks by supporting this kind of reasoning about program structure.

Connections. The symmetric **connect** primitive connects two or more port instances 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

---

[2] Note: the term *subcomponent instance* indicates (dynamic) composition, whereas the term *component subclass* would indicate (static) inheritance.

consistency checks are performed to ensure that each required method is bound to a unique provided method.

A connection also equates shared ownership domains of the same name that appear in connected ports. For example, if the scanner declares **domain** `symbol` in its `out` port, that shared ownership domain will be equivalent to the `symbol` domain in the `in` port of the parser. Thus, any data annotated as owned by the `symbol` domain may be shared between the scanner and parser—and in fact, between these components and the code generator as well, since `symbol` also appears in the parser's `out` port.

Provided methods can be implemented by forwarding invocations to subcomponents or to the required methods of another port. The detailed semantics of method forwarding are given in the language reference manual on the ArchJava web site [Arc02].

ArchJava uses Java's default synchronous, call-and-return semantics for method calls across ports. However, some applications may need alternative connection semantics such as asynchronous, event-based communication. Other work describes an extension to ArchJava that allows developers to define and use connectors with application-specific semantics [ASCN03].

Inheritance. Component classes can inherit from other component classes, or from class `Object`. The compiler's also allows component classes to inherit from ordinary classes (with a warning), at the cost of weakening 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 ports and new provided methods in old ports. However, component subclasses may not specify new required methods because this could break subtype substitutability.

Design Support. ArchJava supports architectural design with **incomplete** components and ports annotated with the keyword, which allow an architect to specify and typecheck an unimplemented architecture specification. That architecture specification can then filled in with code, facilitating a seamless transition between design and implementation.

Communication Integrity. The compiler architecture in Figure 11 shows that while the parser communicates with the scanner and code generator, the scanner and code generator do not directly communicate with each other. Instead, their communication must be mediated through the parser, or through objects in the `symbol` ownership domain that they both share. If the diagram in Figure 11

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. Furthermore, components can only communicate through persistently shared data if the components each declare shared ownership domains that are merged via architectural connections. 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 subcomponents or **unique** components, because this communication may not be declared in the architecture, and thus may violate communication integrity. Nor may a component access data owned by other components, or data that is part of a non-local shared ownership domain. I discuss communication integrity more thoroughly in section 2.4.

2.3.3   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 11 shows the compiler's main method, which creates a **unique** Compiler component and calls its compile method.

Connect Expressions. Dynamically created components can be connected together at run time using a *connect expression*. For instance, Figure 12 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.
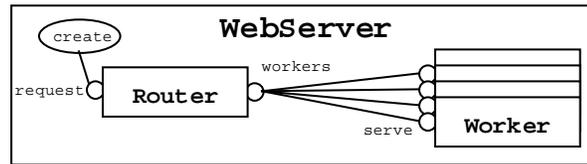
```
public component class WebServer {
  private final owned Router r = new Router();
  connect r.request, create;
  connect pattern Router.workers, Worker.serve;

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

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

public component class Worker extends Thread {
  public port serve {
    group stream;
    provides void httpRequest(stream InputStream in,
                              stream OutputStream out) {
      this.in = in; this.out = out; start();
    }
  }
  public void run() {
    // gets requested file and sends it on the output stream
  }
}
```

Figure 12. 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.

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. Each connect pattern declares a list of (component type, port type) pairs, allowing connections between the ports of subcomponents of the specified type. 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 (or a subtype). 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* is instantiated into a concrete port whenever a connection is declared between the port interface and a port or port interface of a remote component. The created port object is the endpoint for communication through the corresponding connection.

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 are a simple form of dependent type, since the type depends on the value of the instance expression. In order for the type system to track the instance expression soundly, it must be a **final** variable or field—a standard restriction in dependent type systems. Thus, if `r` was not a **final** field, we could not declare the type of `connection` to be `r.workers`.

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`.

Concrete port declarations, such as the ports of the parser in Figure 10, are syntactic sugar for declaring a port interface `p` and a final variable of port interface type **this**.`p`.

Ports are instantiated from port interfaces whenever a connection is made. A connect expression returns a *connection object* that represents the connection. This connection object implements the port interface types of all the connected ports. Thus, in Figure 12, the connect expression implements the interfaces `newWorker.serve` and `r.workers`, and so it is legal to assign the connection object to the `connection` variable, which has type `r.workers`.

Removing Components and Connections. Just as Java does not provide a way to explicitly delete objects, ArchJava does not provide a way to explicitly delete components or connections. Instead, components are garbage collected when they are no longer reachable through direct references, running threads, or architectural connections. Similarly, a connection cannot be disconnected; however, the resources used by the connection will be reclaimed when the connection object is no longer reachable. For example, in Figure 12, 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.

2.3.4   Architectural Style Examples

This subsection shows how ArchJava can express important invariants of two common *architectural styles* discussed by Garlan and Shaw [GS93].

```
public component class PipeAndFilter {
  private final owned Source source = ...;
  private final owned Filter filter = ...;
  private final owned Sink sink = ...;
  connect source.out, filter.in;
  connect filter.out, sink.in;
}

public component class Filter {
 public port in {
    void accept(unique Data d) {
      // process data and send out
      out.accept(process(d));
    }
  }
 public port out {
    requires void accept(unique Data d);
 }
 private unique Data process(unique Data d) {...}
}
```

Figure 13.   A pipe and filter architecture implemented in ArchJava.

Pipe and Filter Architectures.   Figure 13 demonstrates a pipe and filter architecture style, in which the architectural components are filters that accept a stream of data along an input pipe and produce a new stream of data along an output pipe. The PipeAndFilter component class shown is a simple instance of the architectural style with a source, a sink, and a single filter between them. The Filter component accepts data on its input port, processes the data, and sends the new data out its output port.

An important invariant of this architectural style is that the filters do not share state; they communicate only through the pipes connecting them. The alias annotations in the system express and enforce this invariant. Because the Source, Filter, and Sink components share no domains, they cannot directly share any data.[3] The **unique** annotations in the ports express the invariant that when a data structure is passed from one filter to another, the first filter gives up all references to the data.

This example also shows the practical importance of combining uniqueness and ownership in the annotation system. The data passed between components might be a complex data structure that includes multiple internal objects with nontrivial internal aliasing. A type system with only uniqueness could express passing a unique reference to a data structure between components, but could not express the constraint that aliasing is allowed within the data structure but not beyond it. Similarly, a system with only object ownership could express the limited scope of aliasing within the data structure, but could not express the architectural invariant that the first component does not retain any references to the data structure.

---

[3] I am ignoring **shared** annotations, but widespread use of these is poor practice and could be flagged by the compiler.

```
public component class Blackboard {
  private final owned Database store = ...;
  private final owned Client1 c1 = ...;
  private final owned Client2 c2 = ...;
  connect c1.info, store.info;
  connect c2.info, store.info;
}

public component class Database {
  public port interface info {
    domain data;

    requires void notify(lent Message change);

    provides data Data getData(lent Spec spec);
    provides void update(data Data d);
  }
}

public component class Client1 {
  public port info {
    domain data;

    provides void notify(lent Message change);

    requires data Data getData(lent Spec spec);
    requires void update(data Data d);
  }
}
```

Figure 14.   A blackboard architecture expressed in ArchJava.

Blackboard Architectures.   Figure 14 shows a blackboard architectural style, where computational components surround a central data store. The components in the blackboard architecture communicate exclusively by modifying shared state in the data store. Component actions are triggered by changes to the data store made by other components.

In the `Blackboard` component class, the connections show the control flow between the computational components and the data store. These control-flow connections specify that components `c1` and `c2` do not call each other's methods directly, but instead communicate only through method calls to the store—and this specification is verified by ArchJava's type system [ACN02b]. The alias annotations, in turn, describe the data sharing relationships between the components. A glance at the port of the `Database` component shows that the `store`, `c1`, and `c2` components all share the same ownership domain `data`.

The interface of the database shows in more detail how data structures are shared between different parts of the architecture. In its `info` port, the data store defines a **requires** method that it calls to notify clients whenever data has changed. This method passes a change message to the computational components; this message is **lent**, indicating that the clients may not store persistent references to it.

The database also implements two methods allowing clients to get data and to update the store. Here, the specification of what data is requested is a **lent** parameter of `getData`, but the returned data is annotated

```
public component class EventArchitecture {
  private final owned Subject sub = ...;
  private final owned Observer ob = ...;
  connect sub.notify, ob.observe;
}

public component class Observer {
  private owned State state;
  link callback->owned;

  public port observe {
    domain callback;
    requires void register(callback Callback cb);
  }

  public void run() {
    data.register(new MyCallback<owned>(state));
  }
}

public interface Callback {
  public void notify();
}

public class MyCallback<st> implements Callback {
  private st State state;
  public void notify() { state.update(); }
}

public component class Subject {
  private callback Callback cb;

  public port notify {
    domain callback;
    provides void register(callback Callback cb) {
      this.cb = cb;
    }
  }

  public void run() { ... cb.notify(); ... }
}
```

Figure 15. Two components that communicate via a callback object.

with the ownership domain data, indicating that it is shared persistently between different components in the architecture.

Event-based Architectures. Figure 15 shows how an event-based architecture can be defined in ArchJava following the subject-observer pattern and using a callback object for communication [GHJ+94]. The architecture links the notify port of Subject to the observe port of Observer. When the observer starts up, it creates an object of type MyCallback, passing it a reference to the observer's state. This reference is passed to the subject and is stored in an internal field. When an event of interest occurs within the subject, it invokes notify on the callback, which then updates the state of the observer. The connected ports declare the shared ownership domain callback, which is used for the callback object. In

order to allow the callback object to access the internal state of the observer, the `Observer` class declares a link from the `callback` domain to the **owned** domain of the observer.

2.3.5   Summary

ArchJava allows developers to specify the software architecture of a system as a hierarchy of component instances. Connections describe which components within the architecture communicate, and the methods and ownership domains declared in ports show the details of the communication through method calls and shared data. The next section explains in more detail how ArchJava's type system enforces communication integrity, ensuring that the code implementing a system conforms to the architectural specifications.

2.4   Communication Integrity

Communication integrity is the key property enforced by ArchJava, ensuring that communication in the implementation obeys the architectural specification. Intuitively, communication integrity means that components can only communicate with their neighbors in the architecture. In this section, I define communication integrity more precisely, justify the definition, and explain how it is enforced.

2.4.1   Inter-component Communication

Before defining communication integrity, we must define inter-component communication. To do so, we need the concept of an object's *architectural domain*, which can be found by ascending the ownership tree until an ownership domain declared in a component is reached. If an object is **unique**, we assign it a distinguished architectural domain **unique**.

Definition [Inter-component communication]: Two components *communicate* whenever:

1. **Direct call:** Component instance `A` or an object in one of its ownership domains directly accesses (invokes a method or reads or writes a field of) component instance `B`, or

2. **Connection call:** Component instance `A` invokes a method of component instance `B` through a connection, or

3. **Shared data:** An object with architectural domain `A` accesses a non-component object `B`, and `A` and `B` are in different architectural domains.

Java also allows indirect communication via the runtime system (through native methods) and static fields. Formally, the runtime system (including all native methods) and static fields are modeled as part of the ownership domain **shared** that is implicitly declared in every component. Thus, communication through native methods and static methods and fields is treated as shared data communication.

2.4.2 Integrity Definition

Communication integrity in ArchJava is defined as follows:

Definition [Communication Integrity]: All run-time inter-component communication falls into one of the following categories of communication, each of which is documented explicitly or implicitly in the architecture:

1. **Unique communication:** Object A invokes a method on a component instance B that is annotated `unique`, or

2. **Parent-child communication:** Object A invokes a method on a component instance B which is owned by A, or

3. **Connection communication:** Component instance A invokes a method on component instance B through a connection that matches a connect pattern in the component instance that directly owns both A and B, or

4. **Lent communication:** Component instance or object A invokes a method on a non-component object B that has been temporarily lent to A, with either a `lent` annotation or a domain parameter of a method, or

5. **Shared data communication:** Object A accesses some object B in a different architectural domain, and the architectural domains of A and B are linked.

Discussion. Many definitions of communication integrity are possible. I believe that the definition above is a good choice because it allows many different kinds of architectural communication to be expressed, and because it permits only local communication between components.

The first category is communicating with a `unique` component that will be later passed on to another part of the architecture. One typical use of this is when initializing a component in a component factory. Communicating with a `unique` component can be thought of as a special case of parent-child communication (category three), as `unique` refers to a reference that is owned right now but may be transferred in the future. Such communication is also inherently local, since there can be no aliases to a `unique` component.

Parent-child communication could in principle be subsumed under connection communication. However, case studies with ArchJava have shown parent-child communication to be so common in practice that without direct support for it, writing programs would be considerably more awkward. Parents initialize their children by calling their constructors, and they commonly invoke service methods on their children.

This kind of communication is also local, since each component has only one owner, which is the only component permitted to invoke methods directly on it.

Connection communication is the core form of communication supported by ArchJava. It is a natural conceptual idiom, as shown by the innumerable architectural drawings created by system designers everywhere. Furthermore, since connections are only permitted between a parent and its children or between sibling component instances, it is also local.

The fourth category allows a component or object to temporarily lend one of its ownership domains to another component, so that component can perform some action. Conceptually, lending a reference to a domain to another component is like giving up ownership of that domain for the duration of the method call, letting the other component access the objects in the domain, and then getting exclusive ownership back when the method returns. Thus, the ability to lend domain references through method parameters allows temporarily ownership transfers. Treating lent domain parameters as a transfer of ownership is more problematic if components are multithreaded, but in this case an ownership-based locking protocol can be used to ensure exclusive access to an ownership domain [BLR02].

The fifth category, shared data communication, encompasses all forms of communication through shared data—another form of communication that is essential to many systems. In typical programming languages, communication through shared data may be arbitrarily non-local, causing significant problems when understanding and evolving software systems. In ArchJava, shared data communication is local at the architectural level, because each component can only communicate with objects in its domains, and the objects in a domain can only communicate with objects in domains to which it is linked in the architecture.

This definition of communication integrity is not perfect; some aspects of the system (such as the global domain **shared**) give up locality in order to support standard Java idioms like static fields. However, I believe that the definition is a good compromise given the goal of supporting existing Java programs with few changes. Furthermore, I argue that *any* definition of communication integrity that is intended to be general-purpose will have to support the categories of communication described here in some way.

2.4.3 Enforcement

Enforcing communication integrity is essentially ensuring that all instances of inter-component communication fall into one of the architecturally documented categories. Consider the cases of inter-component communication:

1. **Direct call case.** The ArchJava language ensures that if the receiver of a method call is a component, then either the receiver is `this`, or the receiver is `unique` or part of the `owned` domain. In the case of `this`, the communication is within a component. In the cases of `unique` and `owned`, the communication is unique communication and parent-child communication, respectively.

2. **Connection call case.** The type system must ensure that the component that owns both the sender and the receiver declared a connection between them. Whenever a connection is made, the compiler verifies that the components in the connection are owned by the current component, and that the current component declares a connect pattern that matches the components being connected.

3. **Shared data case.** Consider the annotation on the object B being accessed. If the annotation is `unique`, there is no inter-component communication occurring—instead, the calling component is modifying one of its own unique data structures. If the annotation is a locally declared private domain (such as `owned`), there is again no inter-component communication, because the receiver of the access is part of the same component as the sender. If the annotation is `lent` or the domain parameter or a method, the communication is lent communication.

The remaining case is when the accessed object is annotated with an ownership domain that is either declared in a port of the current component or is a domain parameter of the current object. We wish to show that this case is shared data communication. This will be true if and only if architectural domain of the accessing object is linked to architectural domain of the accessed object. But this is guaranteed by the link soundness property, so we are done.

2.4.4 A Relaxed System

The system as presented so far is expressive enough to cover a wide range of programming idioms. However, initial experiences with earlier versions of the system showed that some relaxation may be needed in practice, for at least two reasons [ACN02a]:

Reuse of legacy Java code. Libraries written in Java were not designed with ArchJava-style components in mind, which can create problems when reusing these libraries in an ArchJava program. For example, Java's Swing graphical user interface [ELW98] includes a number of classes that applications extend to build their interface logic. In the system I initially developed component classes could not extend ordinary Java classes, so a developer would not be able to model a Swing-based GUI as part of the architecture.

I chose to allow component classes to extend Java classes and interfaces for the purposes of reusing legacy Java frameworks such as Swing. When reusing legacy Java classes, developers may want to annotate them with ownership parameters. Therefore, the component classes that inherit from the parameterized library classes may also have ownership parameters, even though components do not have ownership parameters in the original ArchJava language.

Communication integrity is relaxed to treat a component's inherited interface as an additional port that is implicitly connected to any object with the capability to access the component. Thus, if one component is owned by another, only objects with a capability to access the parent's owned state can use the inherited interface. Similarly, the ownership parameters of a component allow data sharing with any object that has the right capability, which is more flexible but less structured than sharing objects via ownership domains in ports. Thus, this solution relaxes communication integrity in a controlled way, allowing more flexible access to a component through its inherited interface, but preserving the guarantees for communication via the component's ports and ownership domains. The ArchJava compiler produces a warning message, letting developers know that integrity guarantees are relaxed when components inherit from classes.

Evolution to component code. ArchJava intentionally builds on Java, in part so that developers can express and verify the architecture of existing Java programs using ArchJava. A transition from pure Java to ArchJava is likely to be more effective if programmers can convert one class at a time into a component class [ACN02a]. This is difficult in the system described above, because once one class is converted into a component class, all objects that it communicates with will also have to become component classes so their ports can be linked up to the component in the architecture.

The solution to this problem is to allow ports to be defined in classes as well as components, and to allow connections to link object ports to component ports. That way, when converting a single class into a component class, the component class's ports can be linked to ports in neighboring classes, without fully converting these classes into component classes. Later, the neighboring classes can also be converted into component classes one at a time.

Connections inside a component C can link any object owned by C or one of C's domains to C or any other component or object that C owns. These connections (or connect patterns) look identical to connections between components; the only difference is that an ordinary object (or class) is specified instead of a component instance (or component class).

Connections between a component and an object conceptually allow more communication than connections between components, because an object can be shared more freely than a component. However, they represent a principled relaxation of ordinary, inter-component connections, because although the objects

may be shared according to their alias annotation, communication between the object and the connected component is still declared by the architectural connection.

This solution also supports the idiom where a callback object needs to access the surrounding component. For example, the callback code in Figure 15 could have been written to call a method on the surrounding `Observer` component rather than updating the `State` object directly. The developer simply connects the callback object to the observer component in the observer's architecture. In Java, such callbacks are often implemented as inner classes. Since inner classes have access to the outer class object, ArchJava allows the inner classes of a component to call that component's methods, as if there were an implicit connection between them in the architecture.

With the changes described, I believe that ArchJava is flexible enough to express many Java programs without major implementation changes. The case studies in chapter 4 evaluate this claim.

2.4.5   Summary

Communication integrity means that all communication between components must be declared at the architectural level—either through required and provided methods in connected ports, or through the declaration of an ownership domain in connected ports. The ArchJava compiler enforces communication integrity via local rules governing how references with different alias annotations can be used. Because integrity is enforced through the type system, programmers can develop applications much as they are used to, but gain the assurance that architectural properties are maintained during implementation and evolution.

2.5   ArchJava Implementation

A prototype compiler for ArchJava is publicly available for download at the ArchJava web site [Arc02]. The 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. The compilation technique is modular, so that when a source file is updated, only that file and the files that depend on its interface need to be recompiled. ArchJava's typechecking rules are modular and local, so that programmers can easily identify the cause of a typechecking error.

As of this writing, some of the features of ArchJava are not yet implemented. These include ownership domains (other than the default domain **owned**), design support, and some static and dynamic checks. However, the main alias-control constructs of AliasJava (aside from named domains) and the architectural modeling constructs of ArchJava are all implemented.

The ArchJava compiler translates each component class to an ordinary class with the same name in Java, leaving the fields and method bodies substantially unchanged. Each port interface in the ArchJava source code is compiled into a Java interface containing the required methods of the port interface. An ordinary

port generates the same thing as a port interface as well as an associated field that holds the port instance. All variables of port interface type are compiled to variables of the interface generated for that port interface.

Each connection is compiled into a "connection class" that implements all of the interfaces of the connected ports. The connect expression instantiates this class, passing the connected components to the 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 dissertation use hand-drawn diagrams to communicate architecture; however, I have also constructed a simple visualization tool that generates architectural diagrams automatically from ArchJava source code. I have also begun to develop an IDE for ArchJava using a plug-in for the Eclipse development environment. In the future, I intend to expand this IDE to support graphical browsing and editing of ArchJava architectures, refactoring of Java code to express its architecture, and integration with architectural analysis tools. I also plan to provide other tools such as an `archjavadoc` program that would automatically construct graphical and textual web-based documentation for ArchJava architectures.

Performance. The main cost of the implementation technique, when using standard connections, is that calls through connections are routed through connection objects, adding a layer of indirection to the system. The 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. Casts that involve ownership parameters also involve a small additional overhead, as does the alias type-passing technique. The most significant overhead is incurred by applications that use custom connectors. Since the compiler implements custom connectors by reifying each method call, calling a method across a custom connector is a relatively expensive operation, even if the connector is simple.

Thus far, the only applications of significant size to which I 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]. I expect that most realistic applications would use architectural features at a more coarse grain, and so I anticipate that the time overhead will typically be less than this in practice. I currently have no measurement of the space overhead of ArchJava, but assuming that components and parameterized classes (such as collection classes) tend to be large, I expect

the that space overhead of storing information about ownership parameters and ownership domains will be small in comparison.

2.6 Recent Changes

The ArchJava and AliasJava languages have evolved significantly from their initial presentations [ACN02a,ACN02b,AKC02]. The changes are summarized below.

AliasJava. The major change to AliasJava is the generalization of ownership types to ownership domains. This extension allows developers to more precisely specify aliasing relationships between groups of objects, and also guarantees that inter-component communication through shared data is mediated by ownership domains declared in the architecture.

ArchJava. The most significant change to ArchJava has been the integration of AliasJava's alias control constructs. In the earlier version of ArchJava, the parent of a component was the component that created it. This turned out to be awkward in practice [ACN02b], and so now the owner of a component is its parent. As a result of this change, the system better supports idioms like component factories [GHJ+94], since `unique` components can be created anywhere in the system, then passed to the appropriate place in the architecture where they become owned by their parent component.

2.7 Summary

The ArchJava language extends Java with constructs that model software architecture as a hierarchy of component instances. Components communicate through explicit connections as well as through shared objects that are part of architecturally declared ownership domains. Component communication is mediated through connectors that can have user-defined semantics, even linking components on different machines. ArchJava's type system uses ownership and linearity to enforce structural conformance between architecture and implementation. Thus, engineers can have confidence that the code behaves according to the architectural documentation, and can use this knowledge to build and evolve systems more effectively.

The next chapter formalizes a core subset of the ArchJava language and proves that the type system enforces architectural conformance.

Chapter 3

Formalization

The previous chapter introduced the ArchJava language, and gave an informal argument that its type system enforces integrity. However, Java is a complex language, even without the ArchJava additions. Since Java provides many ways to communicate between components, there is a risk that in designing the ArchJava type system, I may have omitted some communication path that could be used to violate integrity. Thus, we would like some assurance that the informal correctness arguments for ArchJava are valid.

One way to gain increased confidence in the type system is to use formal techniques to model the ArchJava language, and then prove properties about the formal model. 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. Featherweight Java (FJ) formalizes the core of Java, including classes, inheritance, immutable fields, and methods. The rules specifying the static and dynamic semantics of the language are small enough to fit on one page, making it feasible to prove formal properties about the system. Since FJ does not include many features of Java, including field writes, interfaces, inner classes, etc., there is no guarantee that the properties proved of the formal model extend to the full Java language. However, to the extent that the formal system models the most important parts of Java, the proofs increase confidence in the full system.

In this chapter, I formalize the ArchJava as ArchFJ, a core language modeled after Featherweight Java. Because I want to formalize the core constructs of both Java and ArchJava, ArchFJ is considerably more complex than FJ. However, it remains small enough to permit a precise, formal semantics and to permit formal reasoning.

In addition to the Java features modeled by FJ, ArchFJ models core architecture constructs including component classes, port interfaces, connect patterns and connect expressions. It also models the core alias control constructs of AliasJava, including **unique**, **lent**, ownership, and class-level parameterization. Ownership domains are modeled in a simple way: all objects and components have a single domain **owned**, but components can declare additional domains in ports, so that the language can model shared data. In order to reason effectively about uniqueness, ArchFJ models mutable fields instead of FJ's immutable fields.

ArchFJ also makes a number of simplifications relative to ArchJava. Static connections and ports are left out, as they are subsumed by dynamically created connections and port interfaces. As in Featherweight Java, the model omits interfaces, inner classes, and some statement and expression forms, since these

```
CL ::= class C₁<α̅,β̅> extends C₂<α̅> assumes γ->δ,ε->owner { links ε->owned; T̅ f̅; M̅ }
     |   component class K extends E { links α->β; T̅ f̅; M̅ I̅ X̅ }
I  ::= port interface P { domain α̅; R̅ M̅ }
M  ::= T_R m(T̅ x̅) T_this { return e; }
R  ::= requires T_R m(T̅ x̅) T_this;
X  ::= connect pattern K.P̅;

e  ::= vᴬ
     |   new E<p̅>()
     |   e.fᴬ
     |   e.f = e, e
     |   (T)e
     |   e.m(e̅)
     |   θ ▷ e

v  ::= ℓ
     |   x
     |   connect(x̅.P)
     |   null
     |   error

T,V ::= A E<p̅>
     |   v.P
     |   ⋃(v̅.P)
     |   NULL

A,B ::= lent | unique | p
p,q ::= ℓ | α | owned

S  ::= {ℓ → E<ℓ̅>(v̅) | domain(ℓ) }
Γ  ::= {x → T}
Σ  ::= {ℓ → T | domain(ℓ)}
```

Figure 16. ArchFJ Syntax

constructs can be written in terms of more fundamental ones. ArchFJ does not have static fields, so it also omits the **shared** ownership domain, which can be modeled as a domain of a top-level component or object. These omissions make the formal system simple enough to permit effective reasoning, while still capturing the core constructs of ArchJava.

3.1 The ArchFJ Core Language

Syntax. Figure 16 presents the syntax of ArchFJ. The metavariable C ranges over class names; E ranges over component and class names; T ranges over types; K ranges over component class names; f ranges over fields; v ranges over values; e ranges over expressions; P ranges over port interface names; S ranges over stores; ℓ and θ range over locations in the store, where θ is used to represent the value of **this**, and m ranges over method names. Generic alias annotations are represented by A or B, where actual parameters

are sometimes denoted as p or q and formal ownership parameters and domains are named by Greek letters $\alpha, \beta \ldots$. As a shorthand, an overbar is used to represent a sequence.

In ArchFJ, classes are parameterized by a list of alias annotations, and extend another class that has a subsequence of its ownership parameters. An **assumes** clause gives the assumptions about linking between parameters. ArchFJ's classes have only the built-in **owned** domain, which is automatically linked to each ownership parameter of the class. However, classes may declare links from ownership parameters to the **owned** domain as long as corresponding links from the same parameters to the **owner** domain were assumed. Each class defines a set of fields $\bar{f}$ and methods $\bar{M}$; the predefined class Object has no fields or methods. Component classes can extend another component class, or Object. Component classes also define a set of link declarations, port interfaces $\bar{I}$, and connection patterns $\bar{x}$. A port interface is a list of required methods $\bar{R}$, provided methods $\bar{M}$, and domain declarations.

Because we want to reason about communication in the presence of assignment and object identity, ArchFJ adds mutable fields and field assignment to FJ. Therefore, a store S maps locations ℓ to their contents: the class of the object or component, its actual ownership parameters (for ordinary objects) or ownership domains (for components), and the values stored in its fields. I will write S[ℓ] to denote the store entry for ℓ and S[ℓ, i] to denote the value in the ith field of S[ℓ]. Functional store updates at location ℓ are abbreviated S[$\ell \rightarrow$ E$<\bar{\ell}>(\bar{v})$].

Locations are also used to represent ownership domains. The **owned** domain of an object or component is represented by that object's location ℓ. Domains declared within a component are represented by a fresh location, distinct from the component's location. When two ports are connected, the domains declared in the ports become equivalent. Domain equivalence is represented by mapping each domain to its equivalence class representative ($\ell \rightarrow$ **domain**(ℓ_{ecr})), which uniquely represents the set of equivalent domains. I assume a fixed class table *CT* mapping regular and component classes to their definitions. A program, then, is a tuple (*CT*, S, e) of a class table, a store, and an expression.

Expressions include object creation expressions, field reads and writes, casts, and method calls. Several method calls may be executing on the stack at once, and to reason about ownership we will need to know the receiver of each executing call. Therefore, the expression form $\theta \rhd$ e represents a method body e executing with a receiver θ.

Values represent irreducible computational results, and include locations and connections. The **null** value is a distinguished location. ArchFJ represents failed casts and null dereference errors with an explicit

error value. Variables are also considered values because any value (including variables) 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 $\theta \rhd \mathtt{e}$ expressions may appear in the source text of the program; these forms are only generated during reduction. Locations and variables are tagged with their alias type, which otherwise might not be inferable statically. These tags are useful for proving type soundness in the formal system, but they do not affect the run-time semantics of the program and therefore do not exist in the implementation.

In the compiler for the full language, an analysis ensures that each **unique** variable is consumed only once, with all other uses treating that variable as **lent**. ArchFJ models the results of this analysis by explicitly tagging all values with their alias annotation. Thus, a **unique** variable will be annotated **lent** at all of its uses except the consuming use, where it will be annotated **unique**. Similarly, the compiler for the full language performs an analysis to determine that **unique** fields are overwritten immediately after being read. Instead of modeling this analysis formally, ArchFJ provides a destructive read operation (again, identified by the **unique** tag on the field read) that overwrites the field with **null** after every read.

Expressiveness. While ArchFJ has been simplified considerably from ArchJava, it is still quite expressive. For instance, the example architectures in Figures 13-15 can be expressed in ArchFJ with only minor rewriting (e.g., since ArchFJ doesn't include connect statements, we must replace connect statements with pairs of connect patterns and connect expressions).

Types. Ordinary types consist of an alias annotation A and a class name parameterized with annotations p. Annotations may be **lent**, **unique**, **owned**, or a parameter p. Actual ownership parameters in the source text must be ownership parameters α of the enclosing class, or the built-in **owned** domain. However, during reduction, these parameters may be replaced with locations ℓ, indicating the object that corresponds to that actual ownership parameter. Thus, locations are included in the type syntax so that we can give alias types to expressions in an executing program. There is also a type representing **NULL**. Finally, the type system includes port interface types (v.P) and a union type that matches any one of a set of port interface types.

$$\frac{\ell \notin \textit{locations}(\text{S}) \qquad CT(\text{C}) = \textbf{class } \text{C}<\overline{\alpha}> \ldots \qquad \text{S}' = \text{S}[\,\ell \to \text{C}<\overline{\ell}>(\overline{\textbf{null}})\,]}{\text{S}, \theta \vdash \textbf{new } \text{C}<\overline{\ell}>(\,) \to \ell^{\textbf{unique}}, \ \text{S}'} \qquad (\text{R-CNew})$$

$$\frac{\ell, \ell_{1..n} \notin \textit{domain}(\text{S}) \qquad \textit{domains}(\text{K}) = \alpha_{1..n} \qquad \text{S}' = \text{S}[\,\ell \to \text{K}<\ell_{1..n}>(\overline{\textbf{null}})\,, \forall i \in 1..n \ \ell_i \to \textbf{domain}(\ell_i)\,]}{\text{S}, \theta \vdash \textbf{new } \text{K}(\,) \to \ell^{\textbf{unique}}, \ \text{S}'} \qquad (\text{R-KNew})$$

$$\frac{\text{S}[\,\ell\,] = \text{E}<\overline{\ell}>(\overline{\text{v}}) \qquad \textit{fields}(\ell^{\text{A}}, \text{A } \text{E}<\overline{\ell}>) = \overline{\text{T f}} \qquad \text{B} \neq \textbf{unique}}{\text{S}, \theta \vdash \ell^{\text{A}}.\text{f}_i^{\text{B}} \to \text{v}_i^{\text{B}}, \ \text{S}} \qquad (\text{R-Read})$$

$$\frac{\begin{array}{c} \text{S}[\,\ell\,] = \text{E}<\overline{\ell}>(\overline{\text{v}}) \qquad \textit{fields}(\ell^{\text{A}}, \text{A } \text{E}<\overline{\ell}>) = \overline{\text{T f}} \\ \text{S}' = \text{S}[\,\ell \to \text{E}<\overline{\ell}>([\textbf{null}/\text{v}_i]\overline{\text{v}})\,] \end{array}}{\text{S}, \theta \vdash \ell^{\text{A}}.\text{f}_i^{\textbf{unique}} \to \text{v}_i^{\textbf{unique}}, \ \text{S}'} \qquad (\text{R-UniqueRead})$$

$$\frac{\begin{array}{c} \text{S}[\,\ell\,] = \text{E}<\overline{\ell}>(\overline{\text{v}}) \qquad \textit{fields}(\ell^{\text{A}}, \text{A } \text{E}<\overline{\ell}>) = \overline{\text{T f}} \\ \text{S}' = \text{S}[\,\ell \to \text{E}<\overline{\ell}>([\text{v}/\text{v}_i]\overline{\text{v}})\,] \end{array}}{\text{S}, \theta \vdash \ell^{\text{A}}.\text{f}_i = \text{v}^{\text{B}}, \ e \to \ e, \ \text{S}'} \qquad (\text{R-Write})$$

$$\frac{\text{S}[\,\ell\,] = \text{E}<\overline{\ell'}>(\overline{\text{v}}) \qquad \text{A } \text{E}<\overline{\ell'}> \ <: \ \text{A } \text{E}_{\text{C}}<\overline{\ell}>}{\text{S}, \theta \vdash (\text{A } \text{E}_{\text{C}}<\overline{\ell}>)\ell^{\text{A}} \ \to \ \ell^{\text{A}}, \ \text{S}} \qquad (\text{R-Cast})$$

$$\frac{\ell.\text{P} \in \overline{\ell.\text{P}}}{\text{S}, \theta \vdash (\ell.\text{P})\textbf{connect}(\overline{\ell.\text{P}}) \to \ \textbf{connect}(\overline{\ell.\text{P}}), \ \text{S}} \qquad (\text{R-ConnectCast})$$

$$\text{S}, \theta \vdash (\text{T})\textbf{null} \ \to \ \textbf{null}, \ \text{S} \qquad (\text{R-NullCast})$$

$$\frac{\text{S}[\,\ell\,] = \text{E}<\overline{\ell'}>(\overline{\text{v}}) \quad \textit{mbody}(\text{m}, \ell^{\text{A}}, \text{A } \text{E}<\overline{\ell'}>) = (\overline{\text{T x}}, e_0) \quad e_b = [\overline{\text{v}}/\overline{\text{x}}, \ \ell/\textbf{this}]e_0 \quad \text{S}' = \textit{updateD}(\text{S}, e_b)}{\text{S}, \theta \vdash \ell^{\text{A}}.\text{m}(\overline{\text{v}}) \to \ \ell^{\text{A}} \ \triangleright \ e_b, \ \text{S}'} \ (\text{R-Invk})$$

$$\frac{\text{S}[\,\overline{\ell}\,] = \overline{\text{K}<\overline{\ell'}>(\overline{\text{v}'})} \quad \textit{mbody}(\text{m}, \overline{\ell}, \overline{\text{A K}}) = (\overline{\text{T x}}, e_0, i) \quad e_b = [\overline{\text{v}}/\overline{\text{x}}, \ \ell_i/\textbf{this}]e_0 \quad \text{S}' = \textit{updateD}(\text{S}, e_b)}{\text{S}, \theta \vdash \textbf{connect}(\overline{\ell^{\text{A}}.\text{P}}).\text{m}(\overline{\text{v}}) \to \ell_i^{\text{A}} \ \triangleright \ e_b, \ \text{S}'} \ (\text{R-CxtInvk})$$

$$\text{S}, \theta \vdash \ell^{\text{A}} \ \triangleright \ \text{v}^{\text{B}} \to \text{v}^{\text{B}}, \ \text{S} \qquad (\text{R-Context})$$

Figure 17. ArchFJ Evaluation Rules

Reduction Rules. The evaluation relation, defined by the reduction rules given in Figure 17, is of the form $\text{S}, \theta \vdash e \to e', \text{S}'$ read "In the context of store S and receiver θ, expression e reduces to expression e' in one step, producing the new store S'." I write $\to^*$ for the reflexive, transitive closure of $\to$. Most of the rules are standard; the interesting features are how they manipulate architectural constructs and how alias annotations are tracked.

The R-CNew rule reduces an object creation expression to a fresh location tagged as **unique**. The store is extended at that location to refer to a class with the specified ownership parameters, with all **null** fields. The rule for component creation is similar, except that components may not have ownership parameters in the formal system. Instead, the store keeps track of a set of fresh locations representing the domains declared in that component's ports.

There are two rules for field reads. The R-READ rule applies to normal (non-unique) reads of a field f_i; it looks up the receiver in the store and identifies the ith field. The result is the value at field position i in the store. The result value is annotated with the annotation on the field read. The R-UNIQUEREAD rule is similar, but applies to reads annotated as **unique**. Here, the result is always a value with a **unique** annotation, but the value of the field that was read is updated to **null** in the store, written as $[\texttt{null}/v_i]$. This reflects the "destructive read" semantics of the formal language, which models the user-level language's requirement that **unique** fields be updated after unique reads.

The R-WRITE rule is straightforward, updating the ith field of the receiver object with the value written to field f_i. As in Java (and FJ), the R-CAST rule checks that the cast expression is a subtype of the cast type. Note, however, that in ArchFJ this check also verifies that the ownership parameters match, doing an extra run-time check that is not present in Java. Similarly, the rule for a cast to a port interface type verifies that the named port interface type is one of the ones in the actual connection. A cast of a **null** value to any type always succeeds. If the run-time check in the cast rule fails, however, then the cast reduces to the **error** expression (following the cast error rules in Figure 18).

The method invocation rule R-INVK looks up the receiver in the store, then uses the *mbody* helper function (defined in Figure 22) to determine the correct method body to invoke. The method invocation is replaced with the appropriate method body. In the 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 rule for invocations on connections is similar, except that the *mbody* helper function also determines which of the connected components defines the invoked method. For both invocation rules, the store is updated with the *updateD* function, which unifies domains in the store according to connections made in the method body.

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.

$$\frac{S,\theta \vdash e \rightarrow e', \ S'}{S,\theta \vdash e.f_i^{\scriptscriptstyle A} \rightarrow \ e'.f_i^{\scriptscriptstyle A}, \ S'} \qquad\qquad (\text{RC-Read})$$

$$\frac{S,\theta \vdash e \rightarrow e', \ S'}{S,\theta \vdash e.f_i = e_{\text{arg}}, e_{\scriptscriptstyle N} \rightarrow \ e'.f_i = e_{\text{arg}}, e_{\scriptscriptstyle N}, \ S'} \qquad\qquad (\text{RC-RecvWrite})$$

$$\frac{S,\theta \vdash e \rightarrow e', \ S'}{S,\theta \vdash v.f_i = e, e_{\scriptscriptstyle N} \rightarrow \ v.f_i = e', e_{\scriptscriptstyle N}, \ S'} \qquad\qquad (\text{RC-ArgWrite})$$

$$\frac{S,\theta \vdash e \rightarrow e', \ S'}{S,\theta \vdash (T)e \ \rightarrow \ (T)e', \ S'} \qquad\qquad (\text{RC-Cast})$$

$$\frac{S,\theta \vdash e \rightarrow e', S'}{S,\theta \vdash e.m(\overline{e}) \rightarrow e'.m(\overline{e}), \ S'} \qquad\qquad (\text{RC-RecvInvk})$$

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

$$\frac{S,\ell \vdash e \rightarrow e', \ S'}{S,\theta \vdash \ell \rhd e \rightarrow \ell \rhd e', \ S'} \qquad\qquad (\text{RC-Context})$$

$$S,\theta \vdash \textbf{null}.f_i^{\scriptscriptstyle A} \rightarrow \ \textbf{error}, \ S \qquad\qquad (\text{E-ReadNull})$$

$$S,\theta \vdash \textbf{null}.f_i = v, \ e \rightarrow \ \textbf{error}, \ S \qquad\qquad (\text{E-WriteNull})$$

$$S,\theta \vdash \textbf{null}.m(\overline{v}) \rightarrow \ \textbf{error}, \ S \qquad\qquad (\text{E-InvkNull})$$

$$\frac{S[\ell] = E < \overline{\ell'} > (\overline{v}) \qquad A \ E < \overline{\ell'} > \ \not<: \ A \ E_c < \overline{\ell} >}{S,\theta \vdash (A \ E_c < \overline{\ell} >)\ell^{\scriptscriptstyle A} \ \rightarrow \ \textbf{error}, \ S} \qquad\qquad (\text{E-Cast})$$

$$\frac{\ell.P \notin \overline{\ell.P}}{S,\theta \vdash (\ell.P)(\textbf{connect}(\overline{\ell.P})) \rightarrow \ \textbf{error}, \ S} \qquad\qquad (\text{E-ConnectCast})$$

Figure 18. ArchFJ Congruence and Error Rules

Figure 18 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 $\ell \rhd e$ construct: evaluation of the expression e occurs in the context of the receiver ℓ instead of the receiver θ.

The error rules, also in Figure 18, define the semantics of a failed cast or null dereference. Whenever the run-time checks necessary for a cast fail, the cast expression reduces to an **error** value, which is how the system models the exception that is thrown by the full language when a cast fails. Similarly, null dereference errors are modeled by reducing the expression to the **error** value.

$$\frac{CT(\text{E}) \ = \ [\textbf{component}] \ \textbf{class} \ \text{E}<\overline{\alpha},\overline{\beta}> \ \textbf{extends} \ \text{E}'<\overline{\alpha}> \ ...}{\text{E} \ <: \ \text{E}'} \qquad (\text{S{\footnotesize UBTYPE}-C{\footnotesize LASS}})$$

$$\text{T} \ <: \ \text{T} \qquad (\text{S{\footnotesize UBTYPE}-R{\footnotesize EFLEX}})$$

$$\frac{\text{T} \ <: \ \text{T}' \qquad \text{T}' \ <: \ \text{T}''}{\text{T} \ <: \ \text{T}''} \qquad (\text{S{\footnotesize UBTYPE}-T{\footnotesize RANS}})$$

$$\text{v.P} \ <: \ \text{A Object} \qquad (\text{S{\footnotesize UBTYPE}-P{\footnotesize ORT}})$$

$$\textbf{NULL} \ <: \ \text{T} \qquad (\text{S{\footnotesize UBTYPE}-N{\footnotesize ULL}})$$

$$\frac{\text{v.P} \in \overline{\text{v.P}}}{\bigcup(\overline{\text{v.P}}) \ <: \ \text{v.P}} \qquad (\text{S{\footnotesize UBTYPE}-U{\footnotesize NION}})$$

$$\frac{\begin{array}{c}\text{C} \ <: \ \text{C}' \\ \text{A}' = \textbf{lent} \ \vee \ (\text{A} = \text{A}' \ \wedge \ \text{A} \neq \textbf{unique})\end{array}}{\text{A} \ \text{C}<\overline{\text{p}},\overline{\text{q}}> \ <: \ \text{A}' \ \text{C}'<\overline{\text{p}}>} \qquad (\text{S{\footnotesize UBTYPE}-A{\footnotesize LIAS}})$$

$$\frac{\begin{array}{c}\forall \text{p} \in \{\overline{\text{p}}\} \quad (\text{q->p}) \in \mathit{links}(\text{T}) \\ (\text{p}_i\text{->}\textbf{owner}) \in \mathit{links}(\textbf{unique} \ \text{C}<\overline{\text{p}}>) \ \Rightarrow \ (\text{p}_i\text{->q}) \in \mathit{links}(\text{T})\end{array}}{\text{T} \vdash \textbf{unique} \ \text{C}<\overline{\text{p}}> \ <: \ \text{q} \ \text{C}<\overline{\text{p}}>} \qquad (\text{S{\footnotesize UBTYPE}-A{\footnotesize SSIGN}})$$

$$\frac{\text{C} \ <: \ \text{C}' \qquad \{\overline{\text{q}}\} \subseteq \{\overline{\text{p}}\}}{\textbf{unique} \ \text{C}<\overline{\text{p}},\overline{\text{q}}> \ <: \ \textbf{unique} \ \text{C}'<\overline{\text{p}}>} \qquad (\text{S{\footnotesize UBTYPE}-U{\footnotesize NIQUE}})$$

$$\frac{\Sigma(\ell) = \textbf{domain}(\ell_{ecr}) \qquad \Sigma(\ell') = \textbf{domain}(\ell_{ecr})}{\Sigma \vdash \ell = \ell'} \qquad (\text{D{\footnotesize OMAIN}-E{\footnotesize QUALITY}})$$

$$\frac{\forall \ell \in \mathit{domain}(\Sigma) \ \Sigma(\ell) = \text{A} \ \text{E}<\overline{\ell}> \qquad \Sigma'(\ell) = \text{A}' \ \text{E}<\overline{\ell}> \qquad \text{A} \neq \textbf{unique} \Rightarrow \text{A} = \text{A}'}{\Sigma' \ <: \ \Sigma} \qquad (\text{S{\footnotesize UBTYPE}-S{\footnotesize TORE}})$$

Figure 19. ArchFJ Subtyping and Domain Equality

Subtyping Rules. ArchFJ's subtyping rules are given in Figure 19. Subtyping of classes and components is based on the immediate subclass relation given by the **extends** clauses in *CT*. In the S-E{\footnotesize XTENDS} rule and elsewhere, the brackets indicate optional syntax and ellipses indicate syntax that does not affect the rule's semantics. The subtyping relation is reflexive and transitive, and it is required that there be no cycles in the relation (other than self-cycles due to reflexivity). Every type (including port types) is a subtype of Object, **NULL** is a subtype of every type, and a union type is a subtype of all its member types.

The general subtyping rule for types that have alias annotations and parameters follows the class subtyping relation. A type with any annotation can be assigned to a **lent** type, but if the types are given owners then the owners must match. The subtyping rules for **unique** verify that the system preserves the linking constraints in the signature of each class. When a **unique** object becomes part of some domain q, the rule S{\footnotesize UBTYPE}-A{\footnotesize SSIGN} ensures that q is linked to each parameter of the formerly **unique** type, and that all of the assumed links from domain parameters to **owner** are actually present. This rule uses the type of **this**

from the surrounding type judgment; I omit the turnstile when invoking the subtype judgment because the type used is always clear.

The subtyping rule SUBTYPE-UNIQUE for **unique** objects ensures that if one or more parameters are forgotten via subsumption, then all of the forgotten actual domain parameters are still present in the remaining domain parameter list. This guarantees that if the unique object is later assigned to a domain, the test in rule SUBTYPE-ASSIGN will ensure that the owning domain is linked to any hidden parameters of the object as well as to the explicit parameters.

Types are considered equivalent up to equality of domains, defined in rule DOMAIN-EQUALITY; we assume this rule is applied implicitly whenever necessary, using the store typing Σ from the surrounding judgment. A subtyping relation on store types is useful for showing type preservation; one store type is a subtype of another if the subtype has the same type as the supertype for every location in the supertype's domain, except possibly that a domain is substituted for a **unique** annotation.

Typing Rules. Typing judgments, shown in Figure 20, are of the form $\Gamma, \Sigma, T_\theta, \theta \vdash e{:}T$, read, "In the type environment Γ, store typing Σ, receiver class T_θ, receiver instance θ, and set of assumed links *links*, expression e has type T." For a judgment of the form $\Gamma, \Sigma, T_\theta, \theta \vdash e{:}T$ to be well-formed, we require that $\Gamma, \Sigma, T_\theta, \theta \vdash \theta{:}T_\theta$ and any ownership annotations α that appear in T must be bound as parameters or domains of T_θ.

The T-CVAR and T-XVAR rules look up the type of a variable in Γ. The T-LOC rule looks up the type of a location in Σ. Both the variable and location typing rules may assign the expression a supertype of the type in the store type or variable map (for example if the annotation A is **lent** but the store type of the location is **unique**). The object creation rule verifies that any formal arguments assumed to be linked within the instantiated class are actually linked in the current class. The typing rule for **null** assigns it the type **NULL**.

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 T. If the instance expressions in the connection are locations, then there must be a matching connect pattern in their common owner (for simplicity, we omit parent-child connections from ArchFJ).

$$\frac{\Gamma(\text{x}) \ <: \ \text{A} \ \text{E}<\overline{\text{p}}>}{\Gamma,\Sigma,\text{T},\theta \ \vdash \ \text{x}^{\text{A}} \ : \ \text{A} \ \text{E}<\overline{\text{p}}>} \qquad (\text{T-CVar})$$

$$\frac{\Gamma(\text{x}) = \text{v}.\text{P}}{\Gamma,\Sigma,\text{T},\theta \ \vdash \ \text{x} \ : \ \text{v}.\text{P}} \qquad (\text{T-XVar})$$

$$\frac{\Sigma(\ell) \ <: \ \text{A} \ \text{E}<\overline{\ell}>}{\Gamma,\Sigma,\text{T},\theta \ \vdash \ \ell^{\text{A}} \ : \ \text{A} \ \text{E}<\overline{\ell}>} \qquad (\text{T-Loc})$$

$$\frac{\begin{array}{c} CT(\text{E}) = [\textbf{component}] \ \textbf{class} \ \text{E}<\overline{\text{q}}>... \qquad length(\overline{\text{p}}) = length(\overline{\text{q}}) \\ \forall (\text{p}_i \text{->} \text{p}_j) \in links(\textbf{unique} \ \text{E}<\overline{\text{p}}>) \ . \ (\text{p}_i \text{-} \text{p}_j) \in links(\text{T}) \end{array}}{\Gamma,\Sigma,\text{T},\theta \ \vdash \ \textbf{new} \ \text{E}<\overline{\text{p}}>() \ : \ \textbf{unique} \ \text{E}<\overline{\text{p}}>} \qquad (\text{T-New})$$

$$\Gamma,\Sigma,\text{T},\theta \ \vdash \ \textbf{null} \ : \ \textbf{NULL} \qquad (\text{T-Null})$$

$$\frac{\begin{array}{c} \Gamma,\Sigma,\text{T},\theta \ \vdash \ \overline{\text{v}^{\text{A}}} \ : \ \overline{\text{A} \ \text{K}} \\ \overline{\text{v}} = \overline{\text{x}} \Rightarrow (\textbf{connect pattern} \ \overline{\text{K}'.\text{P}} \in connects(\text{T}) \ \wedge \ \overline{\text{A} \ \text{K}} \ <: \ \overline{\text{A} \ \text{K}'} \ \wedge \ \overline{\text{A}} = \overline{\textbf{owned}}) \\ \overline{\text{v}} = \overline{\ell} \Rightarrow (\textbf{connect pattern} \ \overline{\text{K}'.\text{P}} \in connects(\Sigma(\ell)) \ \wedge \ \overline{\text{A} \ \text{K}} \ <: \ \overline{\text{A} \ \text{K}'} \ \wedge \ \ell = owner(\Sigma,\overline{\ell})) \end{array}}{\Gamma,\Sigma,\text{T},\theta \ \vdash \ \textbf{connect}(\overline{\text{v}^{\text{A}}.\text{P}}) \ : \ \bigcup(\overline{\text{v}.\text{P}})} \qquad (\text{T-Connect})$$

$$\frac{\begin{array}{c} \Gamma,\Sigma,\text{T},\theta \ \vdash \ e_0 \ : \ \text{T}_0 \qquad fields(e_0,\text{T}_0) = \overline{\text{T} \ \text{f}} \qquad \text{T}_i \ <: \ \text{A} \ \text{E}<\overline{\text{p}}> \\ (\text{T}_0 = \text{A}_0 \ \text{K}_0) \Rightarrow (meaning(e_0,\theta) = \textbf{this} \vee meaning(\text{A}_0,\theta) = \textbf{owned} \vee \text{A}_0 = \textbf{unique}) \end{array}}{\Gamma,\Sigma,\text{T},\theta \ \vdash \ e_0.\text{f}_i^{\text{A}} \ : \ \text{A} \ \text{E}<\overline{\text{p}}>} \qquad (\text{T-Read})$$

$$\frac{\begin{array}{c} \Gamma,\Sigma,\text{T},\theta \ \vdash \ e_0 \ : \ \text{T}_0 \qquad fields(e_0,\text{T}_0) = \overline{\text{T} \ \text{f}} \qquad \Gamma,\Sigma,\text{T},\theta \ \vdash \ e_1 \ : \ \text{T} \\ \text{T} \ <: \ \text{T}_i \qquad \Gamma,\Sigma,\text{T},\theta \ \vdash \ e_R \ : \ \text{T}_R \\ \text{T}_0 = \text{A} \ \text{K} \Rightarrow (meaning(e_0,\theta) = \textbf{this} \vee meaning(\text{A},\theta) = \textbf{owned} \vee \text{A} = \textbf{unique}) \end{array}}{\Gamma,\Sigma,\text{T},\theta \ \vdash \ e_0.\text{f}_i = e, \ e_R \ : \ \text{T}_R} \qquad (\text{T-Write})$$

$$\frac{\Gamma,\Sigma,\text{T},\theta \ \vdash \ e \ : \ \text{T}_0 \qquad (\text{T}_0 = \text{A} \ \text{E}<\overline{\text{p}}> \ \wedge \ \text{T}_c = \text{A}' \ \text{E}'<\overline{\text{q}}>) \ \Rightarrow \ \text{A} = \text{A}'}{\Gamma,\Sigma,\text{T},\theta \ \vdash \ (\text{T}_c)e \ : \ \text{T}_c} \qquad (\text{T-Cast})$$

$$\frac{\begin{array}{c} \Gamma,\Sigma,\text{T}_{\text{this}},\theta \ \vdash \ e_0 \ : \ \text{T}_0 \qquad mtype(\text{m},e_0,\text{T}_0) = \overline{\text{T}} \times \text{T}'_0 \to \text{T} \\ \Gamma,\Sigma,\text{T}_{\text{this}},\theta \ \vdash \ \overline{e \ : \ \text{T}_a} \qquad \overline{\text{T}_a \ <: \ [\overline{\text{v}/\text{x}}]\text{T}} \qquad \text{T}_0 \ <: \ \text{T}'_0 \\ \text{T}_0 = \text{A} \ \text{K} \Rightarrow (meaning(e_0,\theta) = \textbf{this} \vee meaning(\text{A},\theta) = \textbf{owned} \vee \text{A} = \textbf{unique}) \\ \text{T}_0 = \bigcup(\overline{\text{v}.\text{P}}) \Rightarrow (\textbf{this} \in meaning(\overline{\text{v}},\theta)) \end{array}}{\Gamma,\Sigma,\text{T}_{\text{this}},\theta \ \vdash \ e_0.\text{m}(\overline{e}) \ : \ \text{T}} \qquad (\text{T-Invk})$$

$$\frac{\Gamma,\Sigma,\text{T},\theta \ \vdash \ \ell^{\text{A}} \ : \ \text{T}' \qquad \Gamma,\Sigma,\text{T}',\ell \ \vdash \ e:\text{T}}{\Gamma,\Sigma,\text{T},\theta \ \vdash \ \ell^{\text{A}} \ \rhd \ e \ : \ \text{T}} \qquad (\text{T-Context})$$

Figure 20. ArchFJ Typechecking

The rule for field reads looks up the declared type of the field using the *fields* function defined in Figure 22. Because the read of a **unique** field may be annotated **lent**, the rule checks that the field type is a subtype of the type with the field read annotation substituted for the one in the field type. If the receiver is a component, the receiver must be the current component **this**. Rule T-Write, for field writes, is similar. The cast rule checks that the annotation in the cast expression matches the annotation of the value, because

$$\frac{\mathbf{lent}\ \ldots\notin\overline{T}\qquad \overline{M}\ \text{OK IN}\ E\qquad links(\mathbf{lent}\ E'<\overline{p}>)\subseteq links(\mathbf{lent}\ E<\overline{p}>)}{\mathbf{class}\ E<\overline{p,q}>\ \mathbf{extends}\ E'<\overline{p}>\ \mathbf{assumes}\ \overline{\alpha\text{->}\beta},\varepsilon\text{->}\mathbf{owner}\ \{\mathbf{links}\ \varepsilon\text{->}\mathbf{owned};\ \overline{T\ f};\ \overline{M}\}\ \text{OK}}\quad(\text{T-CLASS})$$

$$\frac{\begin{array}{c}\mathbf{lent}\ \ldots\notin\overline{T}\qquad links(\mathbf{lent}\ E)=\{\overline{\alpha\text{->}\beta}\}\\[4pt](E\ a\ component\ class\ \lor\ E=\text{Object})\qquad \overline{M},\overline{I},\overline{X}\ \text{OK IN}\ K\end{array}}{\mathbf{component\ class}\ K\ \mathbf{extends}\ E\ \{\mathbf{links}\ \overline{\alpha\text{->}\beta};\ \overline{T\ f};\ \overline{M}\ \overline{I}\ \overline{X}\}\ \text{OK}}\quad(\text{T-COMPONENT})$$

$$\frac{\begin{array}{c}CT(E)=[\mathbf{component}]\ \mathbf{class}\ E<\overline{p},\overline{q}>\ \mathbf{extends}\ E'<\overline{p}>\ \ldots\qquad override(m,\ A\ E'<\overline{p}>,\ \overline{T}\times T_{this}\to T)\\[4pt]\{\overline{x:T},\ \mathbf{this}:T_{this}\},\emptyset,T_{this},\mathbf{null}\vdash e\ :\ T_R\qquad T_R\ <:\ T\qquad T_{this}=A\ E<\overline{p,q}>\\[4pt]x:\mathbf{unique}\ E<\overline{p'}>\in\{\overline{x:T},\ \mathbf{this}:T_{this}\}\ \Rightarrow\ x\ occurs\ at\ most\ once\ in\ e\ other\ than\ in\ the\ form\ x^{\mathbf{lent}}\end{array}}{T\ m(\overline{T\ x})\ T_{this}\ \{\ \mathbf{return}\ e;\ \}\ \text{OK in}\ E}\quad(\text{T-METH})$$

$$\frac{\begin{array}{c}CT(K)=\mathbf{component\ class}\ K\ \mathbf{extends}\ E\ \{\overline{T\ f};\ \overline{M'}\ \overline{I}\ \overline{X}\}\qquad \overline{M}\subseteq\overline{M'}\\[4pt]E\ \neq\ \text{Object}\ \Rightarrow\ \mathbf{port\ interface}\ P\ \{\ \overline{\mathbf{domain}\ \alpha};\ \overline{R}\ \overline{M}\ \}\in E\end{array}}{\mathbf{port\ interface}\ P\ \{\ \overline{\mathbf{domain}\ \alpha};\ \overline{R}\ \overline{M}\ \}\ \text{OK in}\ K}\quad(\text{T-PORT})$$

$$\frac{\begin{array}{c}\forall m,i\ \ (mtype(m,\mathbf{this},\mathbf{null}.P_i)=\overline{T}\times\mathbf{lent}\ K_i\to T)\ \Rightarrow\\[4pt](\exists j\neq i\ s.t.\ mtype(m,\mathbf{this},\mathbf{lent}\ K_j)=\overline{T}\times\mathbf{lent}\ K_j\to T\ \land\ \forall k\neq j\ mtype(m,\mathbf{this},\mathbf{lent}\ K_k)\ not\ defined)\end{array}}{\mathbf{connect\ pattern}(\overline{K.P})\ \text{OK IN}\ K}\quad(\text{T-PATTERN})$$

$$\frac{\begin{array}{c}dom(\Sigma)=dom(S)\\[4pt]S[\ell]=E<\overline{\ell}>(\overline{v})\ \Leftrightarrow\ \Sigma(\ell)=A\ E<\overline{\ell}>\\[4pt]S[\ell]=\mathbf{domain}(\ell')\ \Leftrightarrow\ \Sigma(\ell)=\mathbf{domain}(\ell')\\[4pt](S[\ell]=E<\overline{\ell}>(\overline{v})\ \land\ fields(\mathbf{this},\mathbf{lent}\ E<\overline{\ell}>)=\overline{T\ f})\ \Rightarrow\ (\emptyset,\Sigma,\text{Object},\mathbf{null}\vdash\overline{v\ :\ T_v}\land\overline{T_v\ <:\ T})\\[4pt](\Sigma(\ell)=A\ E<\overline{\ell}>(\overline{v})\ \land\ (\ell_1,\ell_2)\in links(A\ E<\overline{\ell}>))\ \Rightarrow\ linked(\Sigma,adomain(\ell_1),adomain(\ell_2))\end{array}}{\Sigma\vdash S}\quad(\text{T-STORE})$$

$$\frac{\begin{array}{c}\Sigma\vdash S\qquad \emptyset,\Sigma,\mathbf{lent}\ \text{Object},\mathbf{null}\vdash e\ :\ T\\[4pt](\Sigma(\ell)=\mathbf{unique}\ E<\overline{\ell}>)\Rightarrow\ell\ occurs\ at\ most\ once\ in\ range(S)\cup e\ other\ than\ in\ the\ form\ \ell^{\mathbf{lent}}\\[4pt]\mathbf{connect}(\overline{\ell.P})\in(range(S)\cup e)\land\mathbf{domain}\ \alpha\in domains(P_i)\land\mathbf{domain}\ \alpha\in domains(P_j)\Rightarrow\\[4pt]ecr(S,lookup(S,\ell_i,\alpha)=ecr(S,lookup(S,\ell_j,\alpha))\end{array}}{\Sigma\vdash(CT,S,e)\ :\ T}\quad(\text{T-MACHINE})$$

Figure 21. Class, Method, Port, Connection, Store, and Machine Typing

annotations cannot be changed via casts (in fact, in the full language, annotations are omitted from casts for this reason).

Rule T-INVK looks up the invoked method's type using the *mtype* function defined in Figure 22, and verifies that the actual argument types are subtypes of the method's argument types. If the invocation is on a component, the component must either be the current component **this**, or be annotated **owned** or **unique**. If the invocation is through a port interface type, 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 receiver class and instance.

Class and Store Typing. Figure 21 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 is OK in E." The class rule checks that none of the field types are **lent**, and ensures that a subclass's linking assumptions are at least as strong than those of its superclass. It also verifies that any methods in the class are well formed. The component class rule ensures that a component only inherits from another component class, or from class Object. It also checks for well-formed ports and connections, in addition to the well-formed method check. Finally, linking declarations in a component must not vary with inheritance.

The rule for methods checks that the method body is well typed, and uses the *override* function (defined in Figure 22) to verify that methods are overridden with a method of the same type. It verifies that **unique** arguments occur at most once in the body of the method other than with annotation **lent**, enforcing the constraint that **unique** values are only consumed once. For component classes, the port typing rule verifies that only subclasses of Object may define new ports, or new required and provided methods within a port. It also ensures that the method signatures in the port are a subset of the methods declared in the class body. The typing rule for connect patterns verifies that for each required method there is a unique provided method with the right signature.

The store typing rule ensures that the store type gives a type to each location in the store's domain that is consistent with the classes and ownership parameters in the actual store. The equivalence class representative for each domain must be consistent in the store and its type. For every value in a field, the type of the value must be a subtype of the declared type of the field. The last two clauses in the store typing rule check that for every pair of domains that are assumed to be linked, either explicitly in an **assumes** clause or based on the implicit link between the owner of an object and that object's parameters, the corresponding architectural domains are linked in the architecture. This condition is required for enforcing integrity.

Finally, the rule T-MACHINE checks that an entire machine configuration is well formed. The first two clauses in the rule check that the expression and store are well typed. The third clause verifies that pointers annotated **unique** in the store type really occur only once in the range of the store together with parts of the executing expression that are annotated **unique** (in addition, the value can occur any number of times in parts of the executing expression that are annotated **lent**). Finally, the fourth clause checks that connected domains are properly unified. For each connect expression in the range of the store or the executing program, for each pair of unified domains, the rule ensures that the equivalence class representatives for the two domains are equal.

Field lookup:

$$fields(e_0, \texttt{Object}) = \bullet$$

$$\frac{\begin{array}{c} CT(\texttt{E}) = [\textbf{component}]\ \textbf{class}\ \texttt{E}\texttt{<}\overline{\texttt{p}},\overline{\texttt{q}}\texttt{>}\ \textbf{extends}\ \texttt{F}\texttt{<}\overline{\texttt{p}}\texttt{>}\ \{\overline{\texttt{T}\ \texttt{f}};\ \ldots\} \\ fields(\texttt{A}\ \texttt{F}\texttt{<}\overline{\texttt{p}'}\texttt{>}) = \overline{\texttt{T}'\ \texttt{f}'} \qquad \overline{\texttt{T}''} = [\overline{\texttt{p}'/\texttt{p}}][\overline{\texttt{q}'/\texttt{q}}][e_0/\textbf{this}][e_0/\textbf{owned}]\ \overline{\texttt{T}} \end{array}}{fields(e_0, \texttt{A}\ \texttt{E}\texttt{<}\overline{\texttt{p}'},\overline{\texttt{q}'}\texttt{>}) = \overline{\texttt{T}'\ \texttt{f}'},\ \overline{\texttt{T}\ \texttt{f}}}$$

Connection lookup:

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

$$\frac{CT(\texttt{K}) = \textbf{component}\ \textbf{class}\ \texttt{K}\ \textbf{extends}\ \texttt{E}\ \{\underline{\textbf{links}}\ \alpha\texttt{->}\beta;\ \overline{\texttt{T}\ \texttt{f}};\ \overline{\texttt{M}}\ \overline{\texttt{I}}\ \overline{\texttt{X}}\} \qquad connects(\texttt{E}) = \overline{\texttt{X}_0}}{connects(\texttt{K}) = \overline{\texttt{X}_0},\ \overline{\texttt{X}}}$$

Method type lookup:

$$\frac{CT(\texttt{E}) = [\textbf{component}]\ \textbf{class}\ \texttt{E}\texttt{<}\overline{\texttt{p}}\texttt{>}\ \ldots\ \{\ldots\overline{\texttt{M}}\ldots\} \qquad \texttt{T}\ \texttt{m}\ (\overline{\texttt{T}\ \texttt{x}})\ \texttt{T}_{\texttt{this}}\ \{\ \textbf{return}\ e;\ \}\in\overline{\texttt{M}}}{mtype(\texttt{m}, e_0, \texttt{A}\ \texttt{E}\texttt{<}\overline{\texttt{p}'}\texttt{>}) = [\overline{\texttt{p}'/\texttt{p}}][e_0/\textbf{owned}][e_0/\textbf{this}]\ \overline{\texttt{T}}\times\texttt{T}_{\texttt{this}} \to \texttt{T}}$$

$$\frac{CT(\texttt{E}) = [\textbf{component}]\ \textbf{class}\ \texttt{E}\texttt{<}\overline{\texttt{p}},\overline{\texttt{q}}\texttt{>}\ \textbf{extends}\ \texttt{E}'\texttt{<}\overline{\texttt{p}}\texttt{>}\ \{\ldots\overline{\texttt{M}}\ldots\} \qquad \texttt{m}\ \textit{is not defined in}\ \overline{\texttt{M}}}{mtype(\texttt{m}, e_0, \texttt{A}\ \texttt{E}\texttt{<}\overline{\texttt{p}'},\overline{\texttt{q}'}\texttt{>}) = mtype(\texttt{m}, e_0, \texttt{A}\ \texttt{E}'\texttt{<}\overline{\texttt{p}'}\texttt{>})}$$

$$\frac{\begin{array}{c} CT(\texttt{E}) = \textbf{component}\ \textbf{class}\ \texttt{K}\ \ldots \qquad domains(\texttt{K}) = \overline{\alpha} \qquad [\Sigma(v) = \texttt{A}\ \texttt{K}\texttt{<}\overline{\ell}\texttt{>}] \\ \textbf{port}\ \texttt{P}\ \{\underline{\textbf{domain}}\ \alpha;\ \overline{\texttt{R}}\ \overline{\texttt{M}}\}\in ports(\texttt{K}) \qquad \textbf{requires}\ \texttt{T}\ \texttt{m}\ (\overline{\texttt{T}\ \texttt{x}})\ \texttt{T}_{\texttt{this}}\in\overline{\texttt{R}} \end{array}}{mtype(\texttt{m}, e_0, v.\texttt{P}) = [\overline{\ell/\alpha}][e_0/\textbf{owned}][e_0/\textbf{this}]\ \overline{\texttt{T}}\times\texttt{T}_{\texttt{this}} \to \texttt{T}}$$

$$\frac{mtype(\texttt{m}, e_0, v_i.\texttt{P}_i) = \overline{\texttt{T}}\times\texttt{T}_{\texttt{this}} \to \texttt{T}}{mtype(\texttt{m}, e_0, \bigcup(\overline{v.\texttt{P}})) = \overline{\texttt{T}}\times\texttt{T}_{\texttt{this}} \to \texttt{T}}$$

Method body lookup:

$$\frac{CT(\texttt{E}) = [\textbf{component}]\ \textbf{class}\ \texttt{E}\texttt{<}\overline{\texttt{p}},\overline{\texttt{q}}\texttt{>}\ \textbf{extends}\ \texttt{F}\texttt{<}\overline{\texttt{p}}\texttt{>}\ \{\ldots\overline{\texttt{M}}\ldots\} \qquad \texttt{T}\ \texttt{m}\ (\overline{\texttt{T}\ \texttt{x}})\ \texttt{T}_{\texttt{this}}\ \{\ \textbf{return}\ e;\ \}\in\overline{\texttt{M}}}{mbody(\texttt{m}, e_0, \texttt{A}\ \texttt{E}\texttt{<}\overline{\texttt{p}'},\overline{\texttt{q}'}\texttt{>}) = [\overline{\texttt{p}'/\texttt{p}},\overline{\texttt{q}'/\texttt{q}}, e_0/\textbf{owned}, e_0/\textbf{this}](\overline{\texttt{T}\ \texttt{x}}, e)}$$

$$\frac{CT(\texttt{E}) = [\textbf{component}]\ \textbf{class}\ \texttt{E}\texttt{<}\overline{\texttt{p}},\overline{\texttt{q}}\texttt{>}\ \textbf{extends}\ \texttt{F}\texttt{<}\overline{\texttt{p}}\texttt{>}\ \{\ldots\overline{\texttt{M}}\ldots\} \qquad \texttt{m}\ \textit{is not defined in}\ \overline{\texttt{M}}}{mbody(\texttt{m}, e_0, \texttt{A}\ \texttt{E}\texttt{<}\overline{\texttt{p}'},\overline{\texttt{q}'}\texttt{>}) = mbody(\texttt{m}, e_0, \texttt{A}\ \texttt{F}\texttt{<}\overline{\texttt{p}'}\texttt{>})}$$

$$\frac{mbody(\texttt{m}, \ell_i, \texttt{A}_i\ \texttt{K}_i\texttt{<}\overline{\ell_i'}\texttt{>}) = (\overline{\texttt{T}\ \texttt{x}}, e)}{mbody(\texttt{m}, \overline{\ell}, \texttt{A}\ \texttt{K}\texttt{<}\overline{\ell'}\texttt{>}) = (\overline{\texttt{T}\ \texttt{x}}, e, i)}$$

Valid method overriding:

$$\frac{mtype(\texttt{m}, \textbf{this}, \texttt{T}_c) = \overline{\texttt{T}'}\times\texttt{A}'\ \texttt{E}\texttt{<}\overline{\texttt{p}}\texttt{>} \to \texttt{T}' \quad\Rightarrow\quad \overline{\texttt{T}'} = \overline{\texttt{T}}\ \wedge\ \texttt{T}' = \texttt{T}\ \wedge\ \texttt{A}' = \texttt{A}}{override(\texttt{m},\ \texttt{T}_c,\ \overline{\texttt{T}}\times\texttt{A}\ \texttt{F}\texttt{<}\overline{\texttt{p}},\overline{\texttt{q}}\texttt{>} \to \texttt{T})}$$

Figure 22. ArchFJ Auxiliary Definitions

Auxiliary Definitions. Most of the auxiliary definitions shown in Figure 22 are straightforward and are derived from FJ. The field and connection lookup rules return the list of fields and connections in a given class. The base case Object has no fields. Field types are translated to the surrounding typing scope by substituting actual parameters for the formal parameters of the receiver, and replacing occurrences of **owned** and **this** in the type with the receiver expression e_0.

Ownership relations:

$$\frac{\Sigma(\ell) = \text{A E<}\overline{\text{p}}\text{>}}{owner(\Sigma, \ell) = \text{A}}$$

$$\frac{\Sigma(\ell) = \text{A K}}{adomain(\Sigma, \ell) = \ell} \qquad \frac{\Sigma(\ell) = \textbf{domain}(\ell')}{adomain(\Sigma, \ell) = \ell} \qquad \frac{\Sigma(\ell) = \textbf{unique C<}\overline{\text{p}}\text{>}}{adomain(\Sigma, \ell) = \textbf{unique}} \qquad \frac{\Sigma(\ell) = \ell' \text{ C<}\overline{\text{p}}\text{>}}{adomain(\Sigma, \ell) = adomain(\Sigma, \ell')}$$

$$linked(\Sigma, \ell, \ell) \qquad linked(\Sigma, \textbf{unique}, \ell) \qquad \frac{\Sigma(\ell'') = \text{A K<}\overline{\ell}\text{>} \qquad \ell -> \ell' \in links(\text{A K<}\overline{\ell}\text{>})}{linked(\Sigma, \ell, \ell')} \qquad \frac{\Sigma(\ell) = \text{A C<}\overline{\ell}\text{>} \qquad \ell' \in \overline{\ell}}{linked(\Sigma, \ell, \ell')}$$

Helper functions:

$$\frac{CT(\text{C}) = \textbf{class C<}\overline{\alpha}, \overline{\beta}\textbf{> extends C'<}\overline{\alpha}\textbf{> assumes } \overline{\gamma -> \delta}, \overline{\varepsilon ->}\textbf{owner} \text{ \{}\overline{\textbf{links } \varepsilon ->\textbf{owned}}; \overline{\text{T f}}; \overline{\text{M}}\text{\}}}{links(\text{A C<}\overline{\text{p}}, \overline{\text{q}}\text{>}) = [\textit{if } \text{A} = \ell \textit{ then } \ell / \textbf{owner}][\overline{\text{p}}/\overline{\alpha}, \overline{\text{q}}/\overline{\beta}] \text{ \{}\overline{\gamma -> \delta}, \overline{\varepsilon ->}\textbf{owner}, \overline{\varepsilon ->}\textbf{owned}\text{\}} \cup \{\textbf{owned}->\gamma, \textbf{owner}->\gamma \mid \gamma \in \{\overline{\alpha}, \overline{\beta}\}\}}$$

$$\frac{CT(\text{K}) = \textbf{component class K extends E \{ links } \overline{\alpha -> \beta}; \text{ ... \}}}{links(\text{A K<}\overline{\ell}\text{>}) = [\overline{\ell}/ groups(\text{K})] \text{ \{}\overline{\alpha -> \beta}\text{\}} \cup \{\textbf{owned}->\gamma \mid \gamma \in groups(\text{K})\}}$$

$$\frac{\text{e} = \theta}{meaning(\text{e}, \theta) = \textbf{this}} \qquad \frac{\text{e} \neq \theta}{meaning(\text{e}, \theta) = \text{e}} \qquad \frac{\text{A} = \theta}{meaning(\text{A}, \theta) = \textbf{owned}} \qquad \frac{\text{A} \neq \theta}{meaning(\text{A}, \theta) = \text{A}}$$

Domain equality and helper functions:

$$\frac{ecr(\text{S}, \ell) = ecr(\text{S}, \ell')}{\text{S} \vdash \ell = \ell'} \qquad \frac{\text{S}[\ell] = \textbf{domain}(\ell')}{ecr(\text{S}, \ell) = \ell'} \qquad \frac{\text{S}[\ell] = \text{K<}\overline{\ell}\text{>}(\overline{\text{v}}) \qquad domains(\text{K}) = \overline{\beta} \qquad \alpha = \beta_i}{lookup(\text{S}, \ell, \alpha) = \ell_i}$$

Store updates:

$$\frac{ecr(\text{S}, lookup(\text{S}, \ell, \alpha)) = \ell_{rep} \qquad ecr(\text{S}, lookup(\text{S}, \ell', \alpha)) = \ell_{other}}{unify(\text{S}, \ell, \ell', \alpha) = \text{S}[\textbf{domain}(\ell_{rep}) / \textbf{domain}(\ell_{other})]}$$

$$\frac{\forall \alpha, i, j \textit{ such that } \textbf{domain } \alpha \in \text{P}_i \ \wedge \ \textbf{domain } \alpha \in \text{P}_j \quad \text{S}' = unify(\text{S}, \ell_i, \ell_j, \alpha)}{updateD(\text{S}, \textbf{connect}(\overline{\ell. \text{P}})) = \text{S}'}$$

$$\frac{\forall \textbf{connect}_i(\overline{\ell_i. \text{P}_i}) \in \text{e} \quad \text{S}_i = updateD(\text{S}_{i-1}, \textbf{connect}_i(\overline{\ell_i. \text{P}_i}))}{updateD(\text{S}_0, \text{e}, \textbf{connect}(\overline{\ell. \text{P}})) = \text{S}_n}$$

Figure 23. More Auxiliary Definitions

ArchFJ follows Java's lookup rules for method types and method bodies, with straightforward extensions for port types and union types. 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 components in a connection defines each method. It then computes the actual method body using the usual *mbody* rule. Both *mbody* and *mtype* translate types to the surrounding typing scope in the same way as *fields*. The expression $[\Sigma(\text{v}) = \text{A K<}\overline{\ell}\text{>}]$ and the corresponding substitution $[\overline{\ell}/\overline{\alpha}]$ in the *mtype* rule for port interface types only apply if the value v is a location (we take the store type Σ from the surrounding typing context). Finally, the *override* rule checks that an overriding method has the same type signature (except for the type of **this**) as the method it overrides.

Figure 23 presents more auxiliary definitions. The *owner* function looks up the owner of a location in the store type—it can return another location or **unique**. The *adomain* function finds the *architectural domain* of a location as follows: the architectural domain of a component or domain is itself, the architectural owner of a **unique** object or component is **unique**, otherwise the architectural domain is defined to be the architectural domain of the object's owner

The *linked* function evaluates whether two domains are linked in the store type. The *linked* relation is reflexive, so each domain is linked to itself (allowing objects in the domain to refer to other objects in the domain). The **unique** architectural domain is linked to all other domains, so that **unique** objects can access objects in other architectural domains. One domain is linked to another if they were both declared by a component that declares the linking relationship explicitly (or implicitly, in the case of **owned**). Also, the **owned** domain of an object is implicitly linked to each of the ownership domain parameters of the object.

The *links* function returns the set of links declared in the **assume** and **links** clauses of a class declaration. It also takes into account the implicit assumption that **owner** and **owned** are linked to each ownership parameter of the class. The function translates formal ownership parameters into actual parameters, and if the alias annotation A on the type is a domain, it replaces **owner** with A. The definition of *links* for component classes is similar, except that explicitly declared links are used instead of assumed links. The *meaning* function allows us to reason about expressions or annotations that are equivalent to **this** or **owned**, even after these variables are replaced with corresponding locations.

In the dynamic semantics, two domains are considered equivalent if they have the same equivalence class representative (a similar rule in Figure 19 applies in the static semantics). The domain equality rule is applied implicitly as needed during reduction. The *ecr* function looks up the equivalence class representative of a domain. The *lookup* function returns the actual domain location representing a particular domain of a component.

Finally, the *updateD* function produces a new store where the equivalence class representatives of domains in a set of connections have been updated to reflect new equalities. For each pair of domains with the same name declared in connected ports, the *unify* rule is invoked. This rule identifies the equivalence class representative for each domain, chooses one arbitrarily, and updates the store replacing the other equivalence class representative with the chosen one.

3.2 Properties

The most important property enforced by ArchFJ is communication integrity. We separate communication integrity into two parts, one for control flow between components, and one for communication through shared data.

The control communication integrity theorem covers direct accesses (calls, field reads and field writes) from one component to another—cases 1, 2, and 3 of the communication integrity definition in section 2.4.2. The theorem states that if a program is well typed and there is an access on a component, then the sender is either that same component (no inter-component communication), or that component's owner (case 2, parent-child communication), or the receiver component is **unique** (case 1, unique communication). Furthermore, if there is a method invocation through a connection, then the sender is one of the connected components, and there is some component that is the owner of all components in the connection, and that owner component declared a connect pattern to which the connection conforms (case 3, connection communication). A more formal statement of the theorem follows:

> **Theorem [Control Communication Integrity]:** If $\Sigma \vdash (CT, \mathtt{S}, \mathtt{e}) : \mathtt{T}$ and $\mathtt{S}, \theta \vdash \mathtt{e} \to \mathtt{e}'$, $\mathtt{S}'$ according to one of the rules R-INVK, R-READ, R-UNIQUEREAD, or R-WRITE, where the typing of the receiver is $\varnothing, \Sigma, \mathtt{T}_\theta, \theta \vdash \ell^A : \mathtt{A}\ \mathtt{K}$, then $\ell = \theta \vee owner(\Sigma, \ell) = \theta \vee owner(\Sigma, \ell) = \mathbf{unique}$. Furthermore, if such a reduction occurs according to the rule R-CXTINVK where the typing of the receiver is $\varnothing, \Sigma, \mathtt{T}_\theta, \theta \vdash \mathbf{connect}(\overline{\ell}.\overline{\mathtt{P}}) : \mathtt{T}$, then $\theta \in \overline{\ell}$ and exists ℓ_o such that $\mathbf{connect}\ \mathbf{pattern}\ \overline{\mathtt{K}'.\mathtt{P}} \in connects(\Sigma(\ell_o))$, and $\forall \ell_i \in \overline{\ell}$, $\ell_o = owner(\Sigma, \ell_i)$ and $\Sigma(\ell_i) <: \mathtt{A}_i\ \mathtt{K}'_i$.

> **Proof:** The ℓ^A case is proved by a case analysis on A. If A=**unique**, then the subtyping test in rule T-LOC implies that $owner(\Sigma, \ell)=\mathbf{unique}$. If A=$\theta$, then by rule T-LOC we have $owner(\Sigma, \ell)=\theta$ or $owner(\Sigma, \ell)=\mathbf{unique}$. Otherwise, rule T-INVK requires that $\ell=\theta$.

> The $\mathbf{connect}(\overline{\ell}.\overline{\mathtt{P}}).\mathtt{m}(\overline{\mathtt{v}})$ case is proved by noting that rule T-INVK checks that $\theta \in \overline{\ell}$, and the remaining conditions are guaranteed by rule T-CONNECT. □

The data communication integrity theorem covers cases 4 and 5 of the integrity definition in section 2.4.2. Consider a well-typed program where there is an access on a non-component object. If the object is

unique this does not represent inter-component communication, because the object is not shared. If the object is **lent** then we have lent communication (case 4).

The interesting case is when the receiver is part of one architectural domain and the sender is part of some other architectural domain (case 5). The data communication integrity theorem states that the architectural domains are linked in the architecture, a property that is verified in the store typing rule. Thus, communication between domains conforms to architectural declarations. More formally:

> **Theorem [Data Communication Integrity]:** If $\Sigma \vdash (CT,\mathtt{S},\mathtt{e}) : \mathtt{T}$ and $\mathtt{S},\theta \vdash \mathtt{e} \rightarrow \mathtt{e'},\mathtt{S'}$ according to one of the rules R-INVK, R-READ, R-UNIQUEREAD, or R-WRITE, where the typing of the receiver is $\varnothing,\Sigma,\mathtt{T}_\theta,\theta \vdash \ell^{\mathtt{A}} : \mathtt{A}\ \mathtt{C}\mathtt{<}\overline{\mathtt{q}}\mathtt{>}$, then either A=**unique**, or A=**lent** or *linked*(Σ,*adomain*(Σ,θ),*adomain*(Σ,A)).

> **Proof:** The cases where ℓ is **unique** or **lent** are trivially valid. In the case where ℓ is part of some domain A, by the conditions implicit in all typing rules, θ has type $\mathtt{T}_\theta$ and A must be bound as a domain or ownership parameter of $\mathtt{T}_\theta$. If A is **owned**, the property is trivially satisfied because **owned** is linked to itself. If $\mathtt{T}_\theta$ is a component type and A is one of its ownership domains, then the component's **owned** domain is linked to A by default, as shown in the definition of *linked*. If A is a parameter of a non-component object, the store typing rule guarantees that the architectural domain of the object is linked to the architectural domain of its parameter. □

Type Soundness. I prove type soundness using standard theorems of type preservation and progress. Type preservation states that if a program is well typed and reduces one step, without resulting in a cast failure or null dereference, the resulting program is also well typed and the resulting expression and store type is a subtype of the previous type.

> **Theorem [Type Preservation]:** If $\Sigma \vdash (CT,\mathtt{S},\mathtt{e}) : \mathtt{T}$ and $\mathtt{S},\theta \vdash \mathtt{e}\rightarrow\mathtt{e'},\mathtt{S'}$ then either $\exists \Sigma' \mathtt{<:} \Sigma, \mathtt{T'} \mathtt{<:} \mathtt{T}$ such that $\Sigma' \vdash (CT,\mathtt{S'},\mathtt{e'}) : \mathtt{T'}$, or else e' has an **error** subexpression.

Before proving type soundness, a lemma is required stating that if a method is well typed, then when actual arguments of the proper types are substituted for formal parameters in the method body, the resulting expression is well typed in the appropriate surrounding context.

Lemma [Term Substitution]: If $\{\overline{\texttt{x:T}}, \texttt{this:}\texttt{T}_{\texttt{this}}\}, \emptyset, \texttt{T}_{\texttt{this}}, \texttt{null} \vdash \texttt{e}_0 : \texttt{T}$, $\Sigma(\ell) = \texttt{A}\ \texttt{E}<\overline{\ell'}>$, $CT(\texttt{E}) = [\texttt{component}]\ \texttt{class}\ \texttt{E}<\overline{\texttt{p}}>...$, and we have type substitution $\Psi = [\overline{\ell'}/\overline{\texttt{p}}, \ell/\texttt{owned}, \ell/\texttt{this}, \overline{\texttt{v}}/\overline{\texttt{x}}]$ such that $\Sigma(\ell)\ \texttt{<:}\ \Psi(\texttt{T}_{\texttt{this}})$, $\emptyset, \Sigma, \texttt{T}_\theta, \theta \vdash \overline{\texttt{v}} : \overline{\texttt{T}_{\texttt{v}}}\ \texttt{<:}\ \Psi(\overline{\texttt{T}})$, and $\texttt{e}_{\texttt{b}} = \Psi(\texttt{e}_0)$, then $\emptyset, \Sigma, \texttt{A}\ \texttt{E}<\overline{\ell'}>, \ell \vdash \texttt{e}_{\texttt{b}} : \texttt{T}_{\texttt{e}}\ \texttt{<:}\ \Psi(\texttt{T})$.

Proof of Lemma: The lemma is proved by induction on the structure of $\texttt{e}_{\texttt{b}}$ with a case analysis on the form of the expression.

Case ℓ: ℓ must be one of the values $\texttt{v}_i$ substituted for one of the variables $\texttt{x}_i$ (including **this**). In this case, we know that $\emptyset, \Sigma, \texttt{T}_\theta, \theta \vdash \texttt{v}_i : \texttt{T}_{\texttt{v}i}\ \texttt{<:}\ \Psi(\texttt{T}_i)$ so the case holds.

Case $\texttt{x}$ is impossible, because all variables in $\texttt{e}_0$ have been substituted with values in $\texttt{e}_{\texttt{b}}$.

Case **null**: Has the type **NULL** as before.

Case **error** is impossible since the original expression $\texttt{e}_0$ and variables $\texttt{v}$ would not be well typed.

Case **new** $\texttt{E}'<\overline{\texttt{q}}>$: This expression will have the same type as before, with actual ownership parameters $\overline{\ell}$ substituted for formal parameters $\overline{\texttt{q}}$.

Case $\texttt{e.f}$: By the induction hypothesis, $\texttt{e}$ is given a subtype of its previous type, modulo ownership parameter substitution. Thus a superset of the original fields will exist, with the same types as before (modulo substitution), and so the whole expression will have the same type (again modulo ownership parameter substitution).

Case $\texttt{e.f=e}_1$, $\texttt{e}_2$: The induction hypothesis allows us to assume that $\texttt{e}$, $\texttt{e}_1$, and $\texttt{e}_2$ are given subtypes of their previous types, modulo ownership parameter substitution. By the argument given above in case $\texttt{e.f}$, the typing of the assignment will go through. Noting that the type of the entire expression is the same as the type of $\texttt{e}_2$, we are done.

Case $(\texttt{T}_{\texttt{C}})\texttt{e}$: By the induction hypothesis, $\texttt{e}$ is well typed. The constraint on equality of alias annotations remains satisfied because the substitution of actual alias annotations for formals is consistent throughout the expression.

Case `connect(v̄.P̄)`: The connect expression will be given the same type as before, except that values will be substituted for any variables in the type. Thus, the tricky part of the case is ensuring that an appropriate connect pattern is declared in the component that owns the connected components. To see that this is true, note that all of the variables in a connect expression must be **owned**, according to the rule T-CONNECT. Since the substituted values must be subtypes modulo the substitution, the substituted values must be owned by the receiver ℓ. Since the receiver's type must be a component type of the form $\Sigma(\ell) = $ A K, by the other checks in rule T-CONNECT we know that the appropriate connect pattern is declared in *connects*(A K) = *connects*($\Sigma(\ell)$). Furthermore, the actual component types in the connection must match those in the connect pattern, again because this is ensured by the check in rule T-CONNECT. This concludes the case.

Case `e.m(ē)`: The assumptions in the lemma and the induction hypothesis are sufficient to ensure that the arguments are well typed and that the method expression is given the same type as before the substitution. The constraints on the type of the receiver are also unaffected by the substitution; for example, *meaning*(**owned**, θ) = **owned** = *meaning*(θ, θ) by design when θ is substituted for **owned**.

Case $\ell^{\text{A}} \rhd$ e is impossible if we only consider expressions that can appear in method bodies. □

Now, the proof of Type Preservation:

Proof of Type Preservation Theorem: By induction on the derivation of S, $\theta \vdash$ e $\rightarrow$ e′, 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-CNEW: We extend the store type to give ℓ the type **unique** C<$\bar{\ell}$> , preserving the type of the resulting expression. The store remains well typed because ℓ is fresh and the values in the fields of ℓ are initialized to **null**. Also, the *neighbor* constraints in the T-STORE rule are satisfied due to the precondition of the T-NEW rule and the assumption that the original store was well typed.

Case R-KNEW: Similar to case R-CNEW. The main difference is that we also add appropriate **domain** constructs to the store type for each instantiated domain.

Case R-READ, R-UNIQUEREAD: Follows from store typing and the rule T-READ. For rule R-UNIQUEREAD, we observe that the unique value is not duplicated since the field being read is overwritten with **null**. Thus, the uniqueness condition in T-MACHINE is preserved.

Case R-WRITE: Follows from store typing and the rule T-WRITE, similar to R-READ. If the annotation B on the right hand side was **unique**, and the field on the left hand side has an owner, we may have to update the owner of the right hand side in the store type to refer to an object instead of to **unique**. However, since the **unique** annotation implies v was previously unique in this case (due to the uniqueness clause in rule T-MACHINE), the store will still be well typed and will have a subtype of its previous typing. If the field is annotated **unique**, we know from the typing rules that the right hand side is annotated **unique** and is the only non-lent occurrence of that location in the system, so the uniqueness condition in rule T-MACHINE is maintained.

Case R-CAST, R-NULLCAST 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 observe that the actual method has the type attributed by *mtype*, modulo substitutions of ownership parameters, **this**, and **owned**. By applying the rule T-INVK, the term-substitution lemma, and the rule for well-typed methods, we see that the substituted method body e_b's type is a subtype of the type returned by *mtype*.

If any of the actual arguments v or the receiver ℓ was **unique**, and the corresponding formal argument is annotated with an owner, we will have to update the store type so that the argument v has the relevant owner. However, since v was previously unique in this case, the store will still be well typed and will have a subtype of its previous typing. The store operation *updateG* ensures that any domains that should be equated due to connections in the method body will have the same equivalence class representative in the store, thus maintaining the condition on domain equivalence in rule T-MACHINE.

We must also ensure that no **unique** arguments are duplicated in the method body, in order to preserve the uniqueness condition in rule T-MACHINE. Here we rely on the rule T-METH, which guarantees that no formal parameter appears in the body of a method more than once with annotation **unique**.

The case is completed by observing that the context construct allows us to type the method body in the context of a new receiver class and instance.

Case R-CXTINVK: Similar to R-INVK, but we extend the induction on *mtype* and *mbody* to cover invocations on connect expressions. The sender and receiver may have different locations representing a shared domain, and so we apply the domain equivalence check in rule T-MACHINE to verify that the locations representing the shared domain have the same equivalence class representative.

Case R-CONTEXT: This case relies on the invariant that the context form can only exist when variables have been substituted with locations within the context expression. This invariant is easy to show, because we do not permit the context form in source code, and the invocation rules that generate the context form do the appropriate substitution of values for variables. In this case, the type of the value in the context expression cannot depend on the receiver class or instance, so the case holds.

Type preservation for the cast error and null dereference error rules follows since these rules reduce to the **error** expression, and it follows trivially for the congruence rules by the induction hypothesis. $\square$

Next, I prove progress, the property that a well-typed program is either a value, or an expression that reduces to another expression via one of the reduction rules.

Theorem [Progress]: If $\Sigma \vdash (CT, \texttt{S}, \texttt{e}) : \texttt{T}$, then either $\texttt{e}$ is an irreducible value, or else $\texttt{S}, \theta \vdash \texttt{e} \rightarrow \texttt{e}', \texttt{S}'$.

Proof: The proof is by induction on the type derivation $\varnothing, \Sigma, \texttt{T}_\theta, \theta \vdash \texttt{e} : \texttt{T}$ for $\texttt{e}$, with a case analysis on the last typing rule used.

Cases T-CVAR, T-XVAR, T-LOC, T-NULL, and T-CONNECT: In all of these base cases, expression $\texttt{e}$ is a value, so the property holds.

Case T-NEW: The typing rule ensures that the right number of ownership parameters is specified, so one of R-CNEW and R-KNEW applies.

Case T-READ: If the receiver is not a value, then RC-READ applies by the induction hypothesis. If the receiver is `null`, E-READNULL applies, resulting in a null pointer error. Otherwise, one of R-READ and R-UNIQUEREAD applies, depending on the whether the read is annotated `unique`, because the typing rule already checked the existence of the relevant field.

Case T-WRITE: If the receiver or the right hand side is not a value, then either RC-RECVWRITE or RC-ARGWRITE applies by the induction hypothesis. If the receiver is null, E-WRITENULL applies, resulting in a null pointer error. Otherwise, R-WRITE applies because the typing rule already checked the existence of the relevant field.

Case T-CAST: If the cast expression is not a value, then RC-CAST applies by the induction hypothesis. Otherwise, all possible cases are covered by the cast reduction and cast error rules.

Case T-INVK: If the receiver or one of the arguments is not a value, then either RC-RECVINVK or RC-ARGINVK applies by the induction hypothesis. If the receiver is null, E-INVKNULL applies, resulting in a null pointer error. If the receiver is a location, simultaneous induction on the operation of *mtype* and *mbody* shows that the check of *mtype* in the typing rule guarantees that *mbody* will return a method body, so rule R-INVK applies.

Finally, if the receiver is a connect expression, we apply a similar induction, but to ensure that a provided method exists matching the required one that was called, we must observe that rule T-CONNECT ensures a matching connect pattern in the architecture, and rule T-PATTERN ensures that a provided method exists for every required method in the connected ports. By this we know that rule R-CXTINVK applies, completing the case.

Case T-CONTEXT: If the right hand side expression is not a value, then RC-CONTEXT applies by the induction hypothesis. Otherwise, R-CONTEXT applies. □

Together, progress and type preservation imply type soundness—a well typed program will not halt unless it computes an irreducible value, or one of two possible run-time errors occur: a null dereference or a failed cast.

Uniqueness and Ownership. The control and data communication integrity theorems each depend on uniqueness and ownership properties. Thus, for communication integrity to be meaningful, we must ensure that **unique** objects really are unique, and that the ownership relation is consistent.

Informally the Uniqueness theorem states that locations annotated **unique** really are unique (except for **lent** references in the currently executing expression). More formally, if a machine configuration is well formed, any location annotated **unique** in the store type occurs at most once either in a field in the store or with a **unique** annotation in the current program expression. Any other occurrences of the location in the current expression must be annotated **lent**.

> **Theorem [Uniqueness]:** If $\Sigma \vdash (CT, S, e) : T$ then
> $(\Sigma(\ell) = \mathbf{unique}\ E<\bar{\ell}>) \Rightarrow \ell$ occurs at most once in range(S) $\cup$ e other than in the form $\ell^{\mathbf{lent}}$.
>
> **Proof:** This condition is checked in rule T-MACHINE. □

The type rules already enforce consistency of ownership annotations within the store and the currently executing program expression. For example, the rule T-STORE ensures that the values stored in an object's fields have types that are compatible with those fields. Similarly, the rule T-LOC checks to ensure that the annotation on the location is compatible with the type of that location in the store type.

Although the type rules enforce consistency in a particular configuration of the abstract machine, they do not ensure that ownership annotations remain consistent over time. The Ownership Soundness theorem states that once a location is given an owner, that owner doesn't change when reduction rules are applied:

> **Theorem [Ownership Soundness]:** If $\Sigma \vdash (CT, S, e) : T$, $\Sigma(\ell) = \ell_o\ E<\bar{\ell}>$, and $S, \theta \vdash e \rightarrow e', S'$, then $\Sigma'(\ell) = \ell_o\ E<\bar{\ell}>$.
>
> **Proof:** This is a corollary of the type preservation theorem above. □

3.3   Summary

This chapter formalized the core of ArchJava as ArchFJ, using a small-step term rewriting semantics. While simple enough to reason formally about, this core is expressive enough to describe realistic architecture examples and common Java implementation idioms. I formally stated a theorem of communication integrity matching the informal definition of communication integrity from chapter 2, and proved that the ArchFJ type system and runtime system enforce it. I proved the standard type soundness property, and stated uniqueness and ownership properties that are corollaries of soundness. Proving properties of the formal model increases confidence in the correctness of the full ArchJava system.

In the next chapter, I take a more practical look at ArchJava, using case studies to evaluate its expressiveness, its usability, and the benefits that it provides to software engineering tasks.

Chapter 4
Evaluation

Previous chapters have introduced the ArchJava language, and have applied a formal model to prove that the core of ArchJava's type system enforces communication integrity. There are many reasons to believe that the technical properties of ArchJava will prove useful, such as the acknowledged importance of developing a good design and adhering to it when building large systems. However, these properties come at a cost, as ArchJava's type system places constraints on the implementation in order to support efficient typechecking. It is important to evaluate this cost, so that researchers and developers can make informed decisions about applying the techniques embodied in ArchJava. Furthermore, as with any new tool, developers will have to learn how to use ArchJava effectively, to realize its potential benefits, and to work around possible limitations.

In order to evaluate the costs and benefits of ArchJava, and to determine how it can be used most effectively, I performed an experiment and two case studies. In all three cases, I began with existing Java code developed by a third party, and (for the case studies) an architecture also specified by the developer. I then attempted to express the system's aliasing patterns and/or architecture using the constructs of ArchJava. This methodology is an effective way to test the expressiveness of ArchJava, because the code and architecture to be expressed is chosen externally, which lessens the danger of choosing the experimental goals to match what ArchJava can express. It also gives one measure of the cost of ArchJava: the additional effort required to retrofit an existing Java program with ArchJava alias and architecture specifications. Finally, the concrete experience of specifying program architectures in ArchJava yields some initial insight into how the language can be used effectively.

The ideal way to explore the benefits of ArchJava would be through a case study tracking a project that uses ArchJava from the design stage, through implementation, and extending to a few significant program evolution tasks as well. This evaluation strategy could confirm the hypothesized benefits of ArchJava in maintaining architectural conformance as a program evolves, and show the costs of the type system as program changes are made. The evaluation in this chapter falls short of this ideal, although one of the case studies includes a few simple evolution tasks. However, this is an important area for future work.

Yet another evaluation strategy would be to conduct a controlled experiment evaluating ArchJava against other alternatives. This strategy would provide more confidence in the observed benefits and costs of ArchJava. However, the nature of ArchJava makes this experiment difficult to design. The hypothesized benefits of architectural conformance show up primarily as a very large project is evolved over time, and it is difficult (and potentially expensive) to perform controlled experiments involving large projects and

significant time scales. Furthermore, the information gleaned from case studies can be much more detailed, even if there is less certainty that the conclusions of the study will generalize to other situations. Because of the cost of experiments, and because exploratory case studies provide more early detail on a system, I have left an experimental evaluation to future work as well.

The case studies described in this chapter took place at varying points in the development of ArchJava. For example, the Aphyds case study was done before the ArchJava compiler supported dynamic architecture or alias annotations—and the study might have been easier if dynamic architectures had been supported. Later, I returned to Aphyds and added alias annotations. All of the case studies were done before I developed the general concept of ownership domains, which allow more precise descriptions of aliasing with only a few additional declarations. While these snapshots do not represent a full evaluation of the final language presented in this document, they have shed considerable light on the properties of ArchJava and have been useful in directing the development of the language.

The outline of this chapter is as follows. First, to determine if ArchJava's alias annotations are expressive enough to describe real Java code, I applied them to `java.util.Hashtable`, one of the more commonly used and complex classes in the Java collections library (Section 4.1). Next, I evaluated the practicality and engineering benefits of ArchJava through two exploratory case studies. The more thorough case study applied ArchJava to Aphyds, a pedagogical circuit-layout application (Section 4.2). A second case study with Taprats, an application for designing Islamic tiling patterns, tested the dynamic architecture support in ArchJava (Section 4.3). I summarize the evaluation in Section 4.4.

4.1 Alias Annotation Expressiveness

Goal. The goal of the study was to address the following experimental questions:

- Can the annotation system effectively express the aliasing invariants of collection class code?

- How much effort is required to annotate existing code?

- Can annotations be done locally, without annotating all transitively reachable code?

Methodology. I evaluated the AliasJava subset of ArchJava by annotating `Hashtable` from the `java.util` collection class library (from the JDK 1.2.1). `Hashtable` is an interesting test case for a number of reasons. It is part of an industrial-strength library with many features and warts. The class must distinguish different ownership domains for the keys, values, and possibly the entries in the `Hashtable`. `Hashtable` is also one of the more complex pieces of the library, so it is a relatively challenging test

case. The Flexible Alias Protection paper used a simplified version of `Hashtable` as a running example in their paper, so this allows a partial comparison to related work [NVP98].

The original source code of `java.util.Hashtable` was 934 lines of code, including comments. I added alias annotations by hand to the `Hashtable` code, attempting to express the aliasing semantics of the code with the simplest and most general annotations possible. The goals of simplicity and generality are sometimes in conflict, and I discuss the tradeoffs in more detail below.

One challenge in this case study was that `Hashtable` depends on a number of other classes in the Java standard library. Thus, in order to typecheck `Hashtable`, the compiler must assume annotations on these other library classes. In this study, I tested a local annotation technique intended to allow the verification of the alias constraints within the `Hashtable` code without annotating the entire Java standard library. I annotated and typechecked `Hashtable` in its entirety, but added only minimal, unchecked annotations to the parts of the standard library used by `Hashtable`. The annotations added to `Hashtable` are then sound if the annotations we added to the standard library are conservative.

Results. I was successful at annotating `Hashtable` with alias types after making one change to the source code (discussed below). In addition to modifying the code for `Hashtable`, partial annotations were added to 17 other classes, including `java.lang.Object`, `ObjectInputStream` and `ObjectOutputStream` from the I/O library, several interfaces and abstract classes in `java.util`, and seven exception classes. In most cases I only had to annotate one or two methods from each external class, suggesting that it is practical to annotate only a local portion of a large system.

The study took about 2 hours and 20 minutes of programming time, not counting occasional interruptions to fix problems with the ArchJava compiler. This is a relatively small investment compared to the time spent developing this library, suggesting that the annotation system is practical for developing new code. However, it would still be time-consuming to add alias annotations to a very large system; a better solution is to infer the annotations automatically, or add annotations incrementally to just the most critical parts of the system.

In return for this time investment, the benefits of the study include documentation of the aliasing constraints of `Hashtable` (for example, it doesn't mix keys and values), and confidence that the implementation correctly preserves these constraints. While these constraints are obvious to anyone familiar with a hash table abstraction, this type of documentation would enable developers to use unfamiliar data structures more effectively.

Several excerpts from the source code highlight lessons learned from the study and suggest potential benefits of AliasJava. For example, I decided to give `Hashtable` three ownership parameters: one each for keys, values, and entries:

```
public class Hashtable<key, value, entry>
            extends Dictionary<key, value>
            implements Map <key, value, entry>,
                       Cloneable,
                       java.io.Serializable { ...
```

The choice of three ownership parameters is a balance between flexibility on the one hand and simplicity and comprehensibility on the other. For example, I could have reduced the number of parameters by merging the `entry` and `key` parameters. On the other hand, I could have added additional parameters also. For example, `Hashtable` has methods for returning the sets of keys, values, and entries. The same set object is returned from these methods each time they are called, and so the type system needs to describe what ownership domain contains the sets. I chose to annotate the `keySet` method's return type as `key Set<key>`, but instead, I could have added extra ownership parameters to `Hashtable` to get a type of `keyset Set<key>`. However, adding three extra ownership parameters to the hash table to represent the key, value, and entry sets would make the class harder to understand and use. Default values for these extra parameters (e.g., `keyset=key`) would alleviate this problem somewhat, but the parameters still add what seems to be unnecessary complexity. This example illustrates that the best alias annotation for a piece of code is not necessarily the most general.

The private inner `Enumerator` class below is part of the original, unannotated code defining an `Iterator` over the keys, values, and entries of the `Hashtable`:

```
  private class Enumerator implements Iterator {
    int type; // KEYS or VALUES or ENTRIES
    public Object nextElement() {
      Entry e = ...;
      return type == KEYS ? e.key :
             (type == VALUES ? e.value : e);
    }
  }
```

The same code is used for keys, values, and entries; the value returned by `nextElement` is determined by the value of the `type` flag. Because I wanted to use separate ownership parameters for keys, values, and entries, I could not give this code a static type as it was. Instead, I converted this code to always return an entry so that I could give it the alias type `entry`. I then defined two wrapper classes that implement `Iterator` and extract and return the key and value from the hash table entry returned by `Enumerator.nextElement`.

The set of `Hashtable` keys is implemented with a simple `KeySet` class that illustrates how inner classes are handled in ArchJava:

```
  private class KeySet extends AbstractSet<key> {
    public unique Iterator<key> iterator() lent {
      return new KeyEnumerator(true);
    }
    // other methods...
  }
```

In this code, class `Keyset` can reference the `key` parameter of the enclosing `Hashtable` class even though `KeySet` has no ownership parameters of its own.

The class `Collections` contains a set of static methods that are used by many of the classes in `java.util`:

```
public class Collections {
  public static unique Set<elements>
    synchronizedSet<elements>(
                        unique Set<elements> s) {
      return new SynchronizedSet(s);
  }
```

The `synchronizedSet` method is used by the `Hashtable` to synchronize access to its key, value, and entry sets. This method shows the need for method parameterization in the AliasJava annotation system: `synchronizedSet` needs to be parameterized by the owner of the elements in the collection so that it can be used to synchronize sets with any element parameter.

The comment for the method above states, "In order to guarantee serial access, it is critical that **all** access to the backing set is accomplished through the returned set." In other words, there should be no lingering aliases to the set passed to this method, because access through these aliases would not be synchronized. The original library did not enforce this constraint; however, I used alias annotations to enforce this constraint by annotating the set argument with **unique**.

Problematic Classes. As described above, I annotated a number of other classes in addition to `Hashtable`; these annotations were not checked by the compiler, but `Hashtable` was checked against the asserted annotations. In general, the annotations we applied to classes other than `Hashtable` were what we would expect to have used if the compiler had been checking those annotations as well. The lone exceptions were certain methods of `ObjectInputStream` and `ObjectOutputStream`. The alias annotations expressed the conceptual semantics of these serialization-related methods (e.g., `writeObject` accepts a **lent** argument and `readObject` returns a **unique** object). This conceptual semantics is incorrect, because the actual implementation of serialization methods stores objects internally, so that if an object is written twice to a stream, the reconstructed object will be read twice. Although it would be nice to express the precise semantics of serialization in AliasJava, the type system is not powerful enough to model this. However, annotating these library methods with unchecked alias types allows us to successfully typecheck clients of these classes.

Summary. My experience annotating `java.util.Hashtable` shows that the alias annotations in ArchJava are able to express some of the aliasing constraints of a small but complex body of existing Java code. To the best of my knowledge, no previous type system supporting ownership-based encapsulation has been evaluated in practice on Java library code. Showing that ArchJava's alias annotations are practical and support reasoning in ordinary Java code is an important first step towards ensuring that the full ArchJava language is practical and beneficial. The next two sections take another step towards this goal by evaluating the architectural features of ArchJava through case studies on real applications.

4.2   Case Study: Aphyds

In order to evaluate the practicality and engineering benefits of ArchJava, I used an exploratory case study to answer the following experimental questions:

- Can ArchJava express the architecture of a real program of significant complexity?

- How difficult is it to reengineer a Java program in order to express its architecture explicitly in ArchJava?

- Does expressing a program's architecture in ArchJava help or hinder software evolution?

4.2.1   Methodology

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

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

I was the participant in the case study—at the time, a graduate student with five year's experience of systems programming in Java. Although I was the developer of the ArchJava compiler, I was unfamiliar with Aphyds and had little experience writing user interfaces in Java. Thus, the study reflects a common reality of a programmer asked to evolve an unfamiliar system.

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

In order to gain insight into ArchJava's support for software evolution tasks, I performed three experiments. First, I analyzed the inter-component communication patterns in Aphyds, describing and categorizing each different message. Next, I refactored the architecture to simplify and regularize these inter-component communication patterns. Finally, I removed a defect from both the original source code and the ArchJava version of Aphyds.

The next two subsections describe the reengineering process and the software evolution experiments. The initial study was done on an earlier version of ArchJava, without alias annotations, so I also report on how the experience affected the ArchJava language design, and on a later expansion of the study to incorporate alias annotations as well.

4.2.2 Reengineering Aphyds

Aphyds, for <u>A</u>cademic <u>P</u>hysical <u>D</u>esign <u>S</u>ystem, is a pedagogical circuit layout application written by an electrical engineering professor for one of his classes. Students are given the program with several key algorithms omitted, and are asked to code the algorithms as assignments. The developer is an experienced programmer with a Ph.D. in computer science, but had no Java background prior to writing Aphyds. The application code is 12,101 lines long, not counting the Java and Symantec libraries used.

Figure 24 shows the developer's drawing of the conceptual architecture of Aphyds on the left, which is re-drawn on the right for clarity. According to the developer, this abstraction allows him to evolve the system even though the code base is too large to hold in his head at once.

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

I decided to test this hypothesis by using the Reflexion Model technique [MNS01] to compare the connections in the developer's conceptual diagram with actual communication patterns between classes in the source code. To each of the developer's conceptual components, I assigned one or more
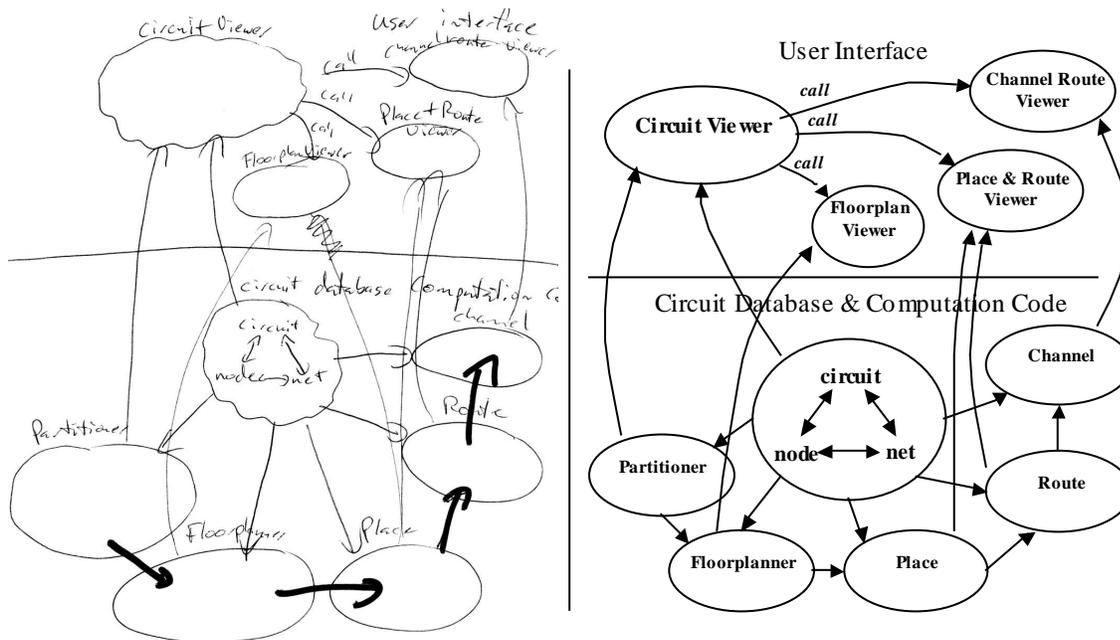
Figure 24. The architecture of Aphyds. On the left is the developer's original drawing, which has been redrawn on the right for clarity. The architecture follows the Model-View design pattern, with the user interface above the line in the middle of the diagram and the circuit database and computational code below. The user interface consists of the `CircuitViewer` window and several subsidiary windows. Below the line are a circuit database of `Node` and `Net` objects and a set of computational modules that act on the circuit database. The unlabeled arrows represent data flow, while the arrows labeled call represent control flow.

implementation classes. I ignored library classes as well as data structures shared by the whole application. I compared the call graph computed by a simple tool to the arrows in the developer's diagram, reversing the direction of his dataflow arrows to reflect control flow in the opposite direction.

Overall, the architecture was a good overview of communication in Aphyds. However, the study revealed several minor missing communication paths in the architecture. For example, although most calls in the application go from the user interface into the model, we found two callbacks going the opposite direction. I also discovered that the communication paths between the CircuitViewer and the other viewer objects were actually bi-directional.

Moreover, this architecture is also incomplete in some important respects. It does not describe the multiplicity or temporal lifetimes of components. It does not show the internal details of components—for example, the CircuitViewer component is made up of several panes and sub-windows that might be of interest to a developer evolving the program. Several complex and messy multi-object communication protocols, dealing with diverse issues, are represented with single lines in the architecture.

Although the developer's conceptual architecture was informal and flawed in certain respects, this is a realistic example of common practice today. Many developers do not define a formal and precise

architecture, but instead communicate the structure of their applications through informal diagrams. One of the motivations for ArchJava is to provide an easy way for developers to gain the benefits of a formal architecture, by embedding it in the code that they write. My experience with the conceptual architecture of Aphyds is summarized by my first hypothesis, which corroborates findings in the Reflexion Model work [MNS01].

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

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

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

I proceeded to reengineer Aphyds to take advantage of the architectural features of ArchJava. My technique was to choose one class at a time from the architectural diagram, and turn it into a component class. I started with the `Circuit` class, as this forms the central part of the architectural diagram. I expected that this process would primarily consist of converting instance variables into ports or subcomponents, invoking methods on ports instead of instance variables, and connecting the ports appropriately in the architecture.

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

I decided to convert the system into a static architecture with components that persisted for the entire execution of the program. My rationale was that this architecture would be simpler to reason about than a dynamically changing architecture. Therefore, I transformed Aphyds to re-initialize old circuit data structures instead of creating new data structures each time the circuit was loaded. I also separated out the

78

refresh logic from the instance-variable setting messages, so that the architectural connections could be set up at startup time, but the display would still work properly throughout program execution. This reengineering process introduced a number of subtle bugs, partly because I did not recognize the dual nature of these messages until partway through the study.

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

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

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

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

I found support for this hypothesis when transforming the remaining classes into components. These smaller tasks went quickly, taking between 30 and 90 minutes each. I spent a total of 30 hours working on Aphyds—15 hours converting the model into components, 8½ hours converting the user interface into components, and 6½ hours refactoring the resulting architecture (as described below). This works out to approximately 2½ hours of work per KLOC. The current code is 12,652 lines long—only 551 lines longer than the original application.

> **Hypothesis 5: Applications can be translated into ArchJava with a modest amount of effort, and without excessive code bloat.**

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

Final Architecture. Figure 25 shows the ArchJava code that expresses the architecture of Aphyds. Although the case study was done before alias annotations were added to ArchJava, I have added them to the example for clarity. Compared to the developer's conceptual architecture, the final ArchJava architecture describes almost identical communication patterns within the circuit database and between the user interface and the database. The multi-way communication between windows that was missing from

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

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

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

  // model components
  final AphydsModel model = ...;

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

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

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

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

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

Figure 25. ArchJava code for the Aphyds and AphydsModel components. There are subcomponent declarations for each element in the user interface, as well as a model component that contains the computational code. Connect declarations show communication patterns between components.

the original architecture but was present in the program has been consolidated into the `window` port of

`Aphyds`.

80
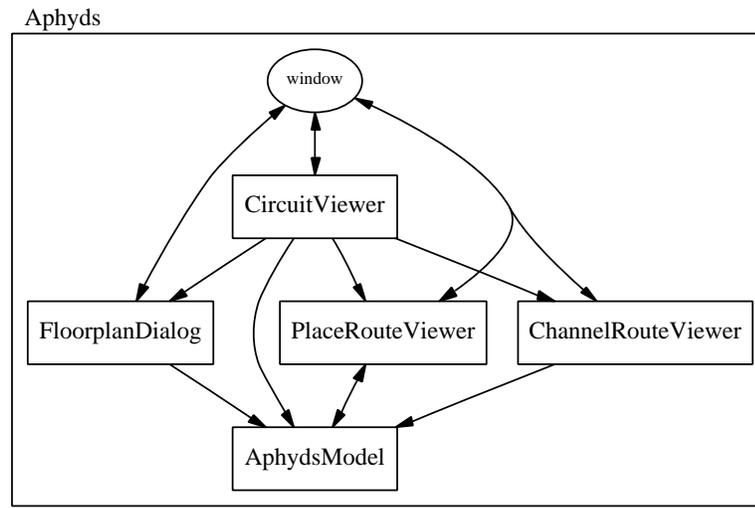
Figure 26. A visualization of Aphyds' architecture, automatically derived from the ArchJava source code. Boxes represent subcomponents, and arrows represent inter-component control flow. The oval denotes the window port, used for window management messages like screen refresh. The circuit database and computational code in the developer's diagram have been isolated in the `AphydsModel` component.

Figure 26 shows a visualization of the current Aphyds architecture, generated automatically from the ArchJava code. The developer of Aphyds examined an earlier version of this diagram, and said that it captures his conceptual architecture well, including the separation between the user interface and the circuit database.

The ArchJava architecture has a number of advantages compared to the original, conceptual architecture. ArchJava architectures are guaranteed to be complete, listing all method call communication between components. The ArchJava architecture is guaranteed to stay up-to-date as the code evolves with changing requirements, and a visualization can be generated automatically. Finally, it is easy to zoom in on an ArchJava architecture to look at the interior structure of a component, determine what methods are in each port, or examine how the methods are implemented.

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

4.2.3   Software Evolution

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

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

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

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

> **Hypothesis 6: Expressing software architecture in ArchJava highlights refactoring opportunities by making communication protocols explicit.**

I decided to systematically analyze the communication patterns to find opportunities for refactoring. For each category of messages, I examined the source code to identify the messages' purpose, the message implementers, the message invokers, and the invocation trigger conditions.

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

Automated tools could have gathered some of this connectivity information from the original Java program. However, these tools would require sophisticated alias analysis to support the level of reasoning about component instances that is provided by ArchJava's communication integrity. Furthermore, ArchJava makes this connectivity explicit at the source code level, and an architect can use ports and connections to express design intent in a way that tools cannot duplicate.

> **Hypothesis 7: Using separate ports and connections to distinguish different protocols and describing protocols with separate provided and required port interfaces may ease program understanding tasks.**

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

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

After this refactoring step, communication in Aphyds was considerably easier to understand. Refactoring eliminated a number of methods and even entire categories of communication. The communication categories in the user interface that remained after refactoring include *menu update*, *window refresh*, and *open/close/show window* messages. Between the user interface and the model, the communication categories were *user interface callback*, *command*, *data query*, *data update*, and *validity check* messages. This experience suggests that the explicitness of architectures in ArchJava may help developers to identify and refactor poorly written code.

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

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

This code traverses a series of object links before calling a method on the final object. It violates a design principle known as the Law of Demeter [LH89], which states, "Objects should only talk to their immediate

neighbors in a system." Code like this makes a program fragile, because this line may break if any object in the sequence of links is changed.

In ArchJava, this code violates communication integrity, because it makes a method call across architectural boundaries without using a connection or a shared ownership domain. Therefore, during the reengineering of Aphyds, I was forced to refactor this code to call a required method on a local port, which was connected through the architecture to the code that enables the menu item.

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

Fixing Defects. Aphyds' developer said that there were subtle defects in the window update code. To investigate how ArchJava affects the defect-fixing process, I identified and removed a defect that was present both in the original Aphyds code and in the ArchJava version. The defect occurred whenever the user changed the location of one element in a routed circuit. The program did not re-compute the routing data, and so the routing display was left in an inconsistent state.

This was a relatively trivial defect, and the solution was the same in both versions: I added a call to the `doGlobalRouting` function from the code that moved the circuit element. I repaired the defect in the ArchJava version first. The repair involved adding a `router` port to the component that moves the circuit element, calling `doGlobalRouting` on that port, and connecting the port to the model in the architecture.

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

```
getDisplayer().placeroutedialog1.placeRouteDisplayer1.getCircuitGlobalRouter()
.doGlobalRouting();
```

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

>**Hypothesis 9: An explicit software architecture can make it easier to identify and evolve the components involved in a change.**

4.2.4   ArchJava Language Changes

While reengineering the Aphyds architecture, I discovered a significant shortcoming in the original ArchJava language design. At first, the language did not include alias annotations to control object aliasing, and so it placed restrictions on control flow from objects into components. For example, objects could not store references to components, and component classes could not inherit from object classes.

These restrictions created a significant problem for framework libraries such as the Swing library [ELW98] used in Aphyds, because these libraries were not written using component classes, yet they must often invoke component methods. This made it impossible to express any meaningful architecture for Aphyds, since all of the application's control flow is driven by the user interface.

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

In order for ArchJava to aid reasoning about communication within the user interface, I decided to also allow component classes to extend regular classes and interfaces, so that legacy libraries could invoke the inherited methods of components through references to the appropriate superclass. Although any ordinary object with access to a component's ownership domain can invoke the inherited methods of that component, the new methods introduced in the component can only be called through declared connections in the architecture. These are the methods that express the application logic that I felt was essential to capture and reason about with software architecture. This solution conveyed the architecture of the user interface much more effectively, and was responsible for a disproportionate amount of the software engineering benefits I observed.

4.2.5   Alias Annotations for Aphyds

After designing and implementing AliasJava, I continued the case study by adding alias annotations to the Aphyds source code, with the goal of answering the following experimental questions:

- Is the annotation system practical on realistic application code?

- Does the annotation system help to encode application-specific architectural constraints?

Methodology. In this study, I focused on the model part of Aphyds. My goal was to express the data sharing relationships between the components in the architecture. Thus, I applied AliasJava to the `AphydsModel` class representing the overall model's architecture, as well as the `Circuit` repository and the five computational module classes. These 7 large classes comprise 3550 lines of code. I typechecked the alias annotations in these classes against annotations I added to parts of the interfaces of the Java standard library and the rest of the Aphyds application.

Results. The study took about three hours and 40 minutes—less than a quarter of the time that it took me to express the control-flow architecture of the same part of Aphyds. The alias annotation system probably required editing more lines of source text than the earlier, control-flow architecture annotations. However, the alias annotations did not require changing any existing source code, just adding annotations. In contrast, expressing the control-flow architecture required significant source-code refactoring to make the code conform to the developer's intended architecture.

I discovered almost immediately that it was quite tedious to annotate the many method arguments (including **this**) and local variable declarations that have a **lent** annotation. I have since made **lent** the default annotation for method arguments and locals.

The annotations in the architecture show the style of sharing in this repository application. The circuit database declares a single ownership domain `data` that represents the circuit elements in the database. Since all of the other computational components act on these circuit elements, they also declare this domain in the ports they use to connect to the database. I did not use the **shared** annotation except for objects of type `String`. `String` objects are immutable in Java, so I did not feel that it was important to track their aliasing patterns precisely, and making strings **shared** simplified the annotation task.

The annotations in ports used for communication between components also show the semantics of the methods used for inter-component communication. Methods that return computed data typically take **lent** parameters and return results annotated either **unique** or `data`. In contrast, methods that set data usually take parameters with `data` annotations. These annotations also showed that the objects shared between components came from a small set of classes including circuit elements and data structures that reflect their organization into a circuit.

4.2.6   Aphyds Case Study Summary

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

4.3   Case Study: Taprats

In this section, I describe a case study that evaluates ArchJava's support for dynamic architectures and component inheritance, and provides another practical evaluation ArchJava. In the case study, I attempt to answer the following experimental questions:

- Is ArchJava expressive enough to describe a real architecture whose precise structure is determined at run time?

- 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?

4.3.1   Methodology

My methodology in this case study was similar to the previous one. I asked the developer to draw the conceptual architecture of Taprats, and then attempted to express the architecture of the program using ArchJava. In the process of the Taprats case study, I refined the hypotheses formed in the Aphyds case study, and made new hypotheses, outlined in bold below.

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.

4.3.2   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 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.
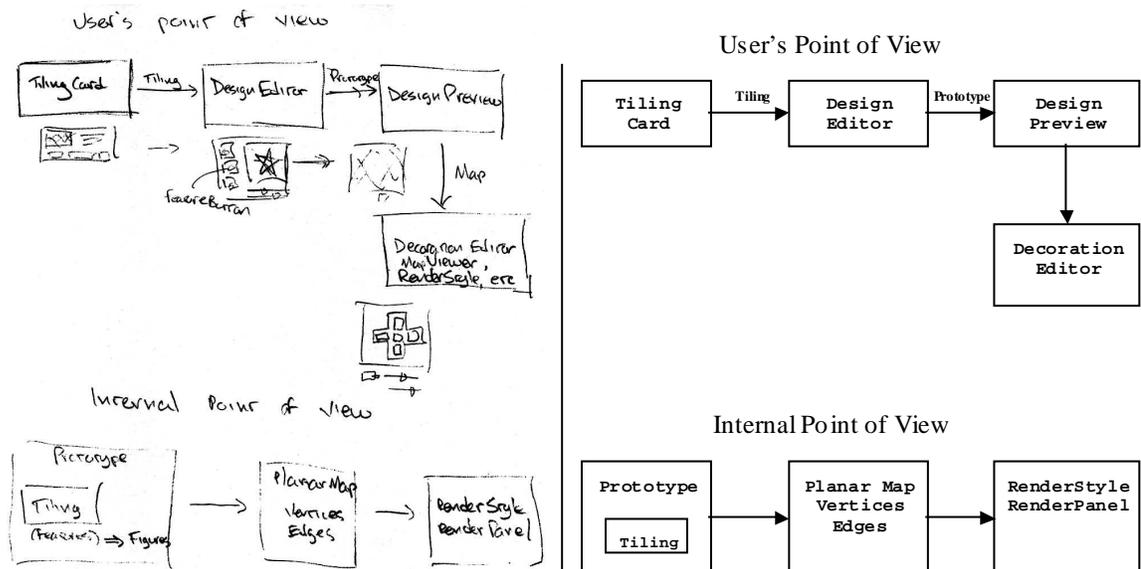
Figure 27. The architecture of Taprats. On the left is the developer's original drawing, which has been redrawn on the right for clarity. At the top of the drawing is the user's point of view, describing the four main user interface windows, what they look like on the screen, and what data structures are passed from one window to the next. At the bottom are the internal data structures, beginning with a `Tiling` that is nested within a `Prototype`, which first evolves into a `Map` and then has rendering style information added.

I asked the developer to draw the conceptual architecture of Taprats. He drew two diagrams, one representing the user interface and one representing the internal data structures. The left side of Figure 27 shows the original diagrams, which are redrawn on the right side of the figure for clarity. 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. I began the study by examining the Taprats source code to try to determine how it corresponds to the developer's conceptual architecture. I 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, my experience with Taprats supports a hypothesis from the previous case study:

Hypothesis 1 (again): 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 data to be shared according to a system of alias annotations, 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, I began the study by creating a component representing the entire Taprats application, and then refined this architecture to increase its level of detail. I developed the following guidelines to 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.

- *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.

- *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. I hope to refine these design principles based on future experience with ArchJava.

Architectural Design. Applying the design principles above, I initially focused on the architecture of the user interface, as shown in the top part of Figure 27. My rationale was that the user interface is the highest level of *scale* in the application.

As I reengineered Taprats, I used the architecture design guidelines to flesh out the initial architecture. Following the developer's conceptual architecture, I made each user interface window into a component. I then refined the architecture by making several window panes into subcomponents of their containing window, either because there was significant *cooperation* between the pane and the window, or because I wanted to document the fact that the panes were *unshared* and they *did not communicate* with other components. Ultimately, I decided not to encode the bottom part of Figure 28 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, I decided to make two structural changes to the application.

First, I made the windows siblings in the architecture instead of being nested within each other. At the time, ArchJava components could only be created by their container component, so I 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 10: 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.

Second, instead of passing tiling data from one window to the next via an argument to the latter window's constructor, I created explicit connections between the windows, along which the data could be passed. I 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.

In response to this concern, I added connection constructors to the language. With these constructors, one window can request a connection to the next, passing all appropriate initialization parameters, and the implementation of the connection constructor in the containing component can create the next window with the appropriate parameters and connect the two directly. This solution expresses the architecture directly, eliminates coupling between windows, and avoids the drawbacks of my original solution.

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

To understand the process of reengineering a program to make its architecture explicit with ArchJava, I recorded the major refactoring steps I 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. This refactoring is probably done more cleanly with the new connection constructor facility, mentioned briefly above, and presented in detail in other work [ASCN03].

- *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, I used several conventional refactoring patterns [FBB+99], as well as a few more minor refactoring patterns that are specific to ArchJava.

Reengineering Cost. I spent about 5½ hours reengineering Taprats, or about 30 minutes of work per thousand lines of code. 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 time was spent actually implementing the architectural changes.

The 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. My experience supports a hypothesis from the previous study:

> **Hypothesis 5 (again): 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 I edited Taprats was that the Taprats code closely followed the Law of Demeter mentioned in the Aphyds case study [LH89]. 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. Because Taprats followed the Law of Demeter, when I converted an object into a component, the new component would often pass the compiler's communication integrity checks as soon as I converted direct method calls into calls on ports. In fact, only one class in Taprats violated the law of Demeter in the source code, and this class was more awkward to componentize compared to other classes in the system. Although the developer had not heard of the Law of Demeter by name, he said that he followed the same

```
public component class Program {

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

  // connections between the windows
  connect pattern TilingSelector.send, DesignEditor.receive {
    send(owned TilingSelector sender, unique Tiling t) {
      owned DesignEditor e = new DesignEditor(t);
      connect(sender.send, e.receive);
    }
  }
  connect pattern DesignEditor.send, PreviewPanel.receive {
    send(owned DesignEditor sender, unique Prototype proto) {
      owned PreviewPanel p = new PreviewPanel(proto);
      connect(sender.send, p.receive);
    }
  }
  connect pattern PreviewPanel.send, RenderPanel.receive {
    send(owned PreviewPanel sender, unique Map m, double left, double top,
        double width, double theta, shared String name) {
      owned RenderPanel r = new RenderPanel(m, left, top, width, theta, name);
      connect(sender.send, r.receive);
    }
  }

  // the main methods of the program
  public void run() {
    owned 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 28. 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`. Connect patterns show communication patterns between windows. Each connect pattern contains a connection constructor which creates and initializes the next window, then connects it to the previous window in the sequence.

principle in his programming, and the code that violated the law of Demeter had been an oversight. This experience suggests:

> **Hypothesis 11: It is 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 28 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, the final ArchJava architecture describes identical communication patterns between the user interface windows. Although the case study was done before alias annotations were added to ArchJava, I have added them to the example for clarity.
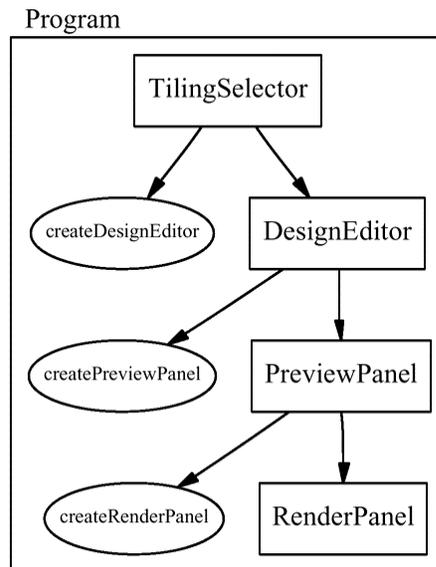
Figure 29. A visualization of Taprats' architecture, automatically derived from the ArchJava source code. Boxes represent subcomponents, and arrows represent inter-component control flow. The ovals are internal ports of the program component, which are used by the first three window components to create the next window in the sequence.

Figure 29 shows a visualization of the Taprats architecture automatically derived from the ArchJava source code using a visualization tool. I showed the developer this diagram, and he agreed that it captured his conceptual architecture well.

Alternative Architectural Choices. The Taprats 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, the architect 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 I 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.

4.3.3   Comparison to Aphyds Case Study

I found that expressing the conceptual architecture of Taprats with ArchJava was straightforward when compared with the earlier Aphyds case study. In all, I spent approximately one fifth the effort in this case

study compared to 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 the original Taprats code was quite similar to the final architecture we expressed in ArchJava. In the Aphyds study, the original code created several components dynamically each time a new file was loaded. I chose to modify the code to reuse old components instead, which may have been a poor choice because it created an architectural mismatch [GAO95] between the original code structure and the final architecture. This required me to restructure the code to support component re-initialization.

- *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.

Experience from the two case studies 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.

4.3.4   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 me 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, I was able to capture the conceptual architecture of Taprats effectively in ArchJava with a small amount of effort relative to the size of the program. This experience demonstrates that the language is flexible enough to describe dynamically evolving software architectures, and suggested improvements to the language design such as connection constructors.

4.4   Summary

I have evaluated the expressiveness an experiment adding alias annotations to a key class from the Java collections library. I evaluated the practicality and the engineering benefits of ArchJava with two case studies on small but real applications: Aphyds and Taprats. The results show that ArchJava is practical enough to document the intended software architecture of existing Java code with a fraction of the effort it takes to write the code in the first place. Furthermore, the case studies suggest that the guaranteed accuracy of ArchJava's architectural documentation provides real benefits for building and evolving software systems.

Chapter 5
Related Work

The ArchJava language builds on diverse fields of related work, including architecture description languages, component infrastructures, module systems, tools for enforcing design, and ownership and linear type systems. ArchJava integrates ideas from many of these areas in order to provide a rich architecture specification language and a practical type system that guarantees architectural conformance.

In the rest of the chapter, I discuss how each of these areas related to ArchJava. At the end, I will summarize the aspects that make the ArchJava project unique.

5.1 Architecture Description Languages

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. The Armani system allows developers to declaratively specify the topological constraints of an architectural style, and then check concrete architectures against that style [Mon01]. 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 support for architectural dynamism is similar to that of Darwin, an ADL designed to support dynamically evolving distributed architectures [MK96].

The SADL system formalizes architectures in terms of theories, providing a framework for proving that communication integrity is maintained when refining an abstract architecture into a concrete one [MQR95]. However, the system did not provide automated support for enforcing communication integrity.

While Wright and SADL are pure design languages, other ADLs have supported implementation in a number of ways. UniCon's tools use an architectural specification to generate connector code that links components together [SDK+95]. C2 provides runtime libraries in C++ and Java that implement C2 connectors [MOR+96]. Darwin provides infrastructure support for implementing distributed systems specified in the Darwin ADL [MK96]. Although the code generation tools are convenient to programmers, they do not automatically enforce communication integrity. Furthermore, these tools support a limited number of built-in connector types, and developers cannot easily define connectors with custom semantics.

Architectures in Rapide can be filled in with implementations in an executable sub-language or in languages such as C++ or Ada. The Rapide system includes a tool that dynamically monitors the execution

of a program, checking for communication integrity violations [Mad96]. The Rapide papers also suggest that integrity could be enforced statically if system implementers follow style guidelines, such as never sharing mutable data between components [LV95]. However, the guideline forbidding shared data prohibits many useful programs, and the guidelines are not enforced automatically.

Component Languages and Infrastructures. A number of recent language proposals add explicit support for components and connections to object-oriented languages such as Java. For example, ComponentJ [SC00] and ACOEL [Sre02] provide primitives for linking components together with connections. However, these languages do not specify architecture explicitly, and thus do not enforce architectural conformance.

Component-based infrastructures such as COM [Mic95], CORBA [OMG95], and Enterprise Java Beans [Sun00] provide sophisticated services such as naming, transactions and distribution for component-based applications. While these infrastructures do not include mechanisms for explicitly describing software architecture, the Arabica environment [RN00] supports C2 architectures built from off the shelf Java Beans components. This system shows how software architecture can be expressed in the context of component infrastructures, but verifying communication integrity of a Java Beans implementation is left to future work.

5.2   Module and Effect Systems

Module systems and 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 [RFS+00]. These tools are derived from research into advanced module systems, exemplified by ML's functors [MTH90] and MzScheme's Units [FF98]. Architecture description languages, including ArchJava, differ from module systems in that the former make data and control flow explicit through architectural connections, while the latter use import/export connections primarily to make names and types defined in one module visible to client modules [MT00].

Compared to ArchJava, advanced module systems have richer facilities for defining, manipulating, and controlling access to types. These facilities support encapsulation, for example by restricting the definition of a type or a function name to within a single module. An important area of future work is combining the strengths of advanced module systems with the architectural conformance property enforced by ArchJava. This combination will be challenging, in part because components are first-class objects that can be created and recursively linked together at run time. First-class, recursive module systems are currently an active area of research [FF98,CHP99,DCH03].

Despite their strengths, existing module systems do not support architectural conformance. For example, if module A defines a function f and restricts its visibility to modules A and B, module C can still call f if module B passes f to C as an anonymous function. This violates architectural conformance if modules A and C are not directly linked with an import/export relationship. Similarly, if module A defines a type T that includes a reference but hides the definition from external modules, and A exposes the reference to external module C, then C can affect variables of type T by writing to the reference. The key issue is that in existing module systems, restricting the visibility of names does not necessarily restrict communication. ArchJava requires all objects (generalizing functions and references) to be labeled with an ownership domain that controls access to those objects. The subtyping rules for unique references ensure that only the architectural neighbors of an ownership domain can access objects in the domain. Thus, in ArchJava, communication between components must be mediated by connections or shared ownership domains, enforcing architectural conformance.

Effect Systems. Effect systems show what functions might be called or what state might be affected by executing a function [LPZ02,CD02]. Effect systems typically show transitive effects in considerable detail, compared to the local, high-level communication overview that an architecture specifies. The additional detail and transitivity that effect systems provide is useful for some engineering tasks, but comes at a cost in verbosity and scalability. For example, ArchJava can summarize communication through a shared callback object using a single connection and ownership domain, whereas an effect system would describe all of the state that could be affected by execution of the callback—possibly a substantial fraction of the entire program. Relative to effects, ArchJava provides a lightweight alternative for describing communication between components that is also less sensitive to program changes that affect effect specifications.

5.3   Enforcing Design

Type Systems. Lam and Rinard have developed a type system for describing and enforcing design [LR03]. Their designs describe communication between subsystems (corresponding to ArchJava's components) that is mediated through shared objects that are labeled with tokens (corresponding to ownership domains). Their system does not model architectural hierarchy, and the set of subsystems and tokens is statically fixed rather than dynamically determined, as in ArchJava. Their system requires whole-program analysis, compared to the local typechecking rules in ArchJava, and it is unclear how their system handles Java features such as inheritance. Furthermore, their system does not describe data sharing as precisely, omitting constructs like uniqueness and encapsulation via ownership. However, they do describe a number of useful analyses, which would complement ArchJava's more detailed architectural descriptions.

Analysis Tools. Design structure can also be supported with analysis. For example, the Reflexion Model system uses a call graph construction analysis in order to find inconsistencies between an architectural model and source code [MNS01]. Similar systems include Virtual Software Classifications [MW99] and Gestalt [SSW96]. These analysis-based approaches are more lightweight than ArchJava's type system, but do not support hierarchical, dynamic architectures or precise data flow constraints.

Aspect-Oriented Programming. Shomrat and Yehudai have proposed using aspect-oriented programming (AOP) to enforce architectural design [SY02]. For example, they show how the constructs of the AOP language AspectJ can be used to enforce kernel architectures, where the kernel of a system has exclusive access to hardware resources and presents a limited interface to the rest of the system. A similar approach [LLW03] has been used to check the Law of Demeter [LH89], a property related to communication integrity. Although aspect-oriented programming gives programmers more control over the properties enforced by the system, the projects described above statically check architectures that are less precise than those supported by ArchJava.

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 [ITU99]. 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 [RSC00]. 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 dissertation can be applied in tools such as Rational Rose RealTime to provide a static guarantee of communication integrity.

Several of CASE tools, including Consystant and Rational Rose RealTime, generate connector code that automatically links distributed components together. This connection code can range from stubs and skeletons for an infrastructure like CORBA or RMI to wires that connect different processors in an embedded system. Like many of the technologies discussed above, these tools typically support a fixed set of connectors, in contrast to the flexibility of user-defined connectors in ArchJava.

5.4   Type Systems for Alias Control

Ownership. Ownership types, which describe a limited static or dynamic scope within which sharing can occur, can also be used to control aliasing. Early work such as Islands [Hog91] and Balloons [Alm97] imposed strict rules on sharing objects between components, significantly limiting expressiveness. A more recent variation, Confined Types [BV99], allows programmers to restrict object references to within a particular package; the system has been extended to support inference of confined types [GPV01]. Universes [MP99] provides a combination of ownership and confinement, providing additional flexibility using read-only references that can cross universe boundaries. More recently, Clarke et al. and Banerjee et al. have used ownership types to reason about side effects and representation independence as well as aliasing [CD02, BN02].

The ownership annotations in AliasJava are most closely related to Flexible Alias Protection [NVP98] and its successors [CPN98,CNP01,Cla01]. Flexible Alias Protection uses ownership polymorphism to strike a balance between guaranteeing aliasing properties and allowing flexible programming idioms. In Flexible Alias Protection, owned objects can only be accessed by their owner and its children. However, this invariant prohibits iterators, which are not owned by a collection, yet must access its owned state. Clarke *et al.* address this issue by introducing a new abstraction called ownership contexts: each object has an *owning context* (the context that owns it) and a *representation context* (the context that owns its representation) [CNP01, Cla01]. The key property of their system is a *containment invariant,* which states that if object o_1 refers to object o_2, then the representation context of o_1 must be *inside* the *owning* context of o_2.

The ownership subset of AliasJava is quite similar to that of Clarke's thesis [Cla01] in both expressiveness and the properties enforced. We wanted to enforce an encapsulation property that relates objects directly, rather than one that relates abstract ownership contexts. Therefore, we chose to phrase the encapsulation guarantees of AliasJava in terms of capabilities that can be passed from one object to another using ownership parameters.   AliasJava's capability-based encapsulation is slightly weaker than Clarke's containment invariant because we place no restrictions on ownership parameters, but AliasJava is correspondingly more flexible. Existing implementations of Flexible Alias Protection and its successors lack support for language features such as inheritance [Bok99, Buc00], and thus there has been no significant experimental validation of the design.

Parameterized Race Free Java (PRFJ) uses the concept of object ownership and uniqueness to develop a type system to guarantee that a program is free of data races [BR01] and deadlocks [BLR02]. PRFJ was not designed to encapsulate owned objects.

Boyapati et al. proposed a system that supports safe run-time updates to code in object-oriented databases [BLS03]. Their system has a stronger notion of object encapsulation than ArchJava: owned objects are confined within the owner, its owned objects, and its inner classes. The system is more restrictive than ArchJava: an object can delegate a capability to access its owned state to its other owned objects and to its inner classes, but not to trusted external classes and methods, even temporarily. Thus, iterators can only be implemented as inner classes of the collection they iterate over. Also, objects cannot be unique if they have non-shared, non-unique ownership parameters—prohibiting many uses of unique. To my knowledge, the system has not been evaluated in practice.

Ownership has also been used to reason about side effects [CD02], representation independence [BN02], and deadlocks and race conditions [BR01,BLR02]. Clarke et al. used the concept of ownership (without explicit annotations) to enforce some of the confinement rules in the JavaBeans specification [CRN03]. Leino et al. use the related concept of data groups to describe different sections of an object's state for the purposes of specifying effects [LPZ02].

Information Flow. Another area of related work is systems that enforce the secure flow of information. A representative system is JFlow [Mye99], which annotates each piece of data with a set of principals that *own* the data, and for each owner, a list of principals that are allowed to *read* the data. The type system verifies that no principal can read a piece of data unless all the data's owners have given read permission to that principal. AliasJava is more lightweight than JFlow, because AliasJava labels references with a single owner instead of a list of owners and a list of authorized readers for each owner. However, AliasJava only supports reasoning about direct information flow between components, not transitive flows from one component to another.

Tools for Understanding Aliasing. An alternative to using a type system to limit aliases is to use an alias analysis-based tool such as Lackwit [OJ97] to visualize the aliases within a program. For answering questions about aliasing, AliasJava can be more precise than Lackwit, which does not treat data structures polymorphically. Compared to Lackwit's successor Ajax [OCa00], AliasJava allows more parametric polymorphism on methods, but its treatment of subtype polymorphism is less precise due to the constraints of AliasJava's type system. One benefit of expressing alias information in a type system is that the information is constantly available and constantly checked for consistency, and so there is no need to run a tool to take advantage of it.

Uniqueness. Uniqueness types can be used to declare references that are unaliased [Min96, CBS98]. Passing a unique object from one method to another avoids all aliasing problems, since the original method may not use the object again. Boyland's alias burying paper [Boy01] described how to implement unique pointers without a special destructive read operation, an innovation adopted by AliasJava. Alias burying uses an effect system to enforce a stronger uniqueness invariant than AliasJava enforces: namely that when a unique field is read, all previous lent aliases to that field are dead. Recently, Clarke and Wrigstad proposed external uniqueness, allowing internal pointers to a unique object as long as only one external pointer is present [CW03]. External uniqueness could be added orthogonally to AliasJava, but I have not yet done so because the making external uniqueness sound in the presence of threads is an open problem.

Linearity. Linear type systems [Wad90a] guarantee uniqueness and in addition can be used to track resource usage. AliasJava's `lent` annotation, which allows temporary aliasing of a unique pointer, is similar to the `let!` construct in Wadler's system [Wad90a]. Linear types have been applied to check protocols defining the order in which library methods can be called, as in the Vault language [FD02]. Leino et al. have also used uniqueness to specify and check side effects in a modular way [LPZ02]. A number of research efforts have used linear types to verify the correctness of explicit memory management using the concept of a region [TT94,CWM99,FD02,GMJ+02]. A region represents a group of objects that are deallocated together. A region type is similar to an ownership type in that all objects must be accessed through their region. Although supporting explicit deallocation is not a goal of AliasJava, the system makes two contributions relative to region types. First, regions must be tracked linearly to enable explicit deallocation; AliasJava relaxes this constraint on owning objects, permitting more flexible aliasing patterns. Second, region types do not have an encapsulation model like AliasJava's for protecting access to the objects in a region; any object that can name the region can access the objects inside it.

Monads. Pure functional languages use monads to achieve linearity when modeling state [W90b], serving a similar purpose to ArchJava's unique qualifier. Ownership domains can be viewed as a mechanism for reasoning about state that sits somewhere between monads and full-blown references: more flexible than the former yet more structured than the latter. Software architecture's role extends beyond reasoning about state, however—evolving any program, functional or stateful, requires understanding what functionality each part of the system implements, and how the system's components work together to accomplish some task.

Other Type Systems. Capabilities for Sharing [BNR01] describes a general capability-based aliasing model that can encode a number of other alias-control systems, including ours, as a special case. The capabilities in their system are fine-grained and are dynamically checked; in contrast, AliasJava verifies statically (except for casts) that objects are only accessed through appropriate high-level capabilities.

Systems such as Alias Types [WM00] and Role Analysis [KLR02] specify the shape of a local object graph in more detail than AliasJava. The Alias Types proposal uses this information to safely deallocate objects, while Role Analysis is used to specify and check properties of data structures. In contrast to these detailed specifications of a local alias graph, the goal of AliasJava is to provide a lightweight and practical way to constrain global aliasing within a program.

Separation logic is an alternative way to designate separate parts of the heap and reason about how they may refer to one another [Rey02]. Different parts of the heap in separation logic are similar to ownership domains in AliasJava. Although separation logic provides a much more detailed way to describe aliasing, AliasJava's constructs are more lightweight, allowing developers to specify heap separation properties with just a few type annotations.

5.5   Summary

ArchJava builds on a great deal of previous work in software architecture, modules, infrastructure software, and type systems. Three factors, taken together, set ArchJava apart from all previous systems:

- A rich specification of software architecture that is hierarchical, instance-based, dynamically evolving, and includes detailed specifications of connector semantics and aliasing constraints;

- A type system integrating uniqueness and ownership, and a formal proof that the core of the type system ensures that all run-time communication follows the architectural specifications; and

- An implementation in a mainstream programming language and numerous case studies on non-trivial programs, showing that the system is practical and provides significant engineering benefits.

Chapter 6
Critique of the ArchJava Project

In this chapter, I critique the ArchJava project: what worked well and what worked poorly in the ArchJava language design and experimental evaluation, and some lessons I hope to apply in future research.

6.1 Language Design

In general, I believe the ArchJava language design achieves its goals. Modeling architecture as a hierarchy of component instances is very natural, and many existing ADLs model architecture in this way [MT00]. AliasJava's use of ownership domains for sharing data between components appears to be novel, but it is a natural object-oriented generalization of architectural shared variables, which were part of the SADL language [MQR95].

Embedding architecture into an implementation language and enforcing architectural structure using types is controversial, but I have shown that the technique offers benefits that no previous technique has. ArchJava is the first system that supports a rich architectural model and enforces architectural conformance in a general-purpose implementation language. The type system is demonstrably practical. Although using ArchJava clearly requires an investment of more effort than analysis-based architectural tools [MNS01], it also lets architects specify the architecture of a system in much more detail, and its presence in the source code provides developers with a constant awareness of architectural issues. Further experience will indicate whether this tradeoff is worthwhile.

Building on top of Java provided great benefits, but also significant drawbacks. The depth of practical experience I got with ArchJava is almost wholly attributable to the ability to leverage existing Java programs. Java was also a convenient vehicle for explaining what I had done, and makes the system potentially adoptable by practitioners. On the other hand, using Java made a lot of things harder and uglier. The worst example is the dichotomy between the component world and the object world—two different kinds of entities with different rules. Although I intend to continue working in mainstream languages, I also look forward to examining how to build a system that uses a unified construct for architectural modeling and object-oriented data modeling.

The practical experience with Java programs has also enabled me to iron out many of the kinks in the language design. For example, ArchJava's support for components inheriting from objects, objects connecting to components, and components requesting connections to their peers in an architecture, were all features that developed in response to needs identified in case studies.

The biggest remaining problem in the language design is the lack of scalability in the architecture. Although the programs in my case studies are nontrivial, they are small enough that their architectures are fairly trivial. Moving to larger, more complex architectures is likely to require hierarchy in ports and connections, which is not currently supported. One mistake in the current language design is that port interfaces are defined within components, instead of being defined externally. Rectifying this will be crucial to supporting larger systems, where port interfaces are reused in multiple places. Finally, case studies have already shown that the language has inadequate support for "glue" connections, which bind the external port of a component to one or more ports of its subcomponents.

6.2   Experimental Evaluation

ArchJava is a real, robust system that has been used in tens of thousands of lines of code. I have performed a number of exploratory case studies on real, nontrivial programs, including additional studies not presented here [ASCN03, Ric02]. The results of the case studies have been rich, and in some cases, surprising.

Despite these successes, the experimental evaluation of ArchJava has had significant limitations, which must be corrected before claims of practicality and benefits can be fully substantiated. The experience I have gathered has been on relatively small systems of up to about 10,000 lines of code. However, I would expect the primary benefits of architecture to accrue in programs of over 100,000 lines of code. Furthermore, I was the subject in all of the case studies described in chapter 2, and none of the case studies provided much information about how ArchJava programs evolve over time.

6.3   Lessons Learned

In addition to the detailed lessons implicit in the criticism above, I learned some higher-level lessons about programming language and software-engineering research, from which I hope to benefit in the future.

Lesson 1: Focus on technical properties early on in the design of a system.

Answering the questions, "What properties should the system have?" and, "Why are those properties useful?" was crucial not only for getting the design right, but also for communicating that design to others. For example, an earlier focus on communication integrity in the overall ArchJava design, as well as a focus on the encapsulation properties enforced by AliasJava, would have enabled me to progress faster.

Lesson 2: Gather experience in depth before breadth.

Many language design papers are published with little in-depth experience—and, although these systems may have technical merit, this lack of practical evaluation raises serious questions about the true benefits and costs of the language designs. I think the Aphyds case study is a good start, combining a realistic (if

106

small) program with evaluation in some depth. However, I wish I had invested the time to do a longitudinal case study on a large program, perhaps reducing the breadth of the ArchJava project (omitting the work on connectors [ASCN03]) in order to get more depth of experience.

Lesson 3: Don't hesitate to tackle controversial research, if there's an achievable path to success.

Risky, controversial research projects don't always work out. ArchJava was controversial in two constituent communities: many programming language researchers didn't see the value of software architecture, while many software architecture researchers thought that embedding architecture into a language was a dead-end approach. However, even though ArchJava is still contentious in some circles, there are signs that type-based approaches to architectural conformance are becoming a topic of interest in the broader research community [LR03,CRN03].

Controversial projects offer risks, but also the potential of significant rewards. In the case of ArchJava, I found that because type systems had not been previously applied to architecture, there was an opportunity to statically enforce an important property, architectural conformance, for the first time. This strategy of applying technical results from one area to an important unsolved problem in another is one potential route to making a broad impact. Time will tell if ArchJava achieves the latter.

Chapter 7
Conclusion

The ArchJava language extends Java with constructs that model hierarchical, dynamically evolving software architectures. Components communicate through explicit connections as well as through shared objects that are part of architecturally declared ownership domains. ArchJava's type system uses ownership and linearity to enforce structural conformance between architecture and implementation. Thus, engineers can have confidence that the code behaves according to the architectural documentation, and can use this knowledge to build and evolve systems more effectively.

I have evaluated the practicality of ArchJava with two exploratory case studies on real systems of nontrivial size. These studies suggest that ArchJava is practical enough to be used on existing systems with relatively minor changes to the code, and that the language provides concrete benefits for software evolution tasks.

In future work, I intend to improve the set of ArchJava development tools so that I can gather experience from outside users of ArchJava. I will perform further case studies to see if the language can be successfully applied to programs larger than 100,000 lines of code. I will also investigate extending the language design to enable reasoning about other architectural properties, such as enforcing an architectural style, checking temporal ordering constraints on component methods, and specifying and checking domain-specific architectural properties. Finally, I will use what I have learned with ArchJava to create other languages and tools that enforce architectural conformance in new domains.

References

[ACN02a] Jonathan Aldrich, Craig Chambers, and David Notkin. ArchJava: Connecting Software Architecture to Implementation. Proc. International Conference on Software Engineering, Orlando, Florida, May 2002.

[ACN02b] Jonathan Aldrich, Craig Chambers, and David Notkin. Architectural Reasoning in ArchJava. Proc. European Conference on Object-Oriented Programming, Málaga, Spain, June 2002.

[ADG98] Robert Allen, Remi Douence, and David Garlan. Specifying and Analyzing Dynamic Software Architectures. Proc. Fundamental Approaches to Software Engineering, Lisbon, Portual, March 1998.

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

[AKC02] Jonathan Aldrich, Valentin Kostadinov, and Craig Chambers. Alias Annotations for Program Understanding. Proc. Object Oriented Programming Systems, Languages and Applications, Seattle, Washington, November 2002.

[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.

[Alm97] Paulo Sérgio Almeida. Balloon Types: Controlling Sharing of State in Data Types, Proc. European Conference on Object-Oriented Programming, Jyväskylä, Finland, June 1997.

[ASCN03] Jonathan Aldrich, Vibha Sazawal, Craig Chambers, and David Notkin. Language Support for Connector Abstractions. Proc. European Conference on Object-Oriented Programming, Darmstadt, Germany, July 2003.

[BHL94] Edoardo Biagioni, Robert Harper, and Peter Lee. Implementing Software Architectures in Standard ML. Proc. ICSE-17 Workshop on Research Issues in the Intersection of Software Engineering and Programming Languages, 1994.

[BLR02] Chandrasekhar Boyapati, Robert Lee, and Martin Rinard. Ownership Types for Safe Programming: Preventing Data Races and Deadlocks. Proc. Object-Oriented Programming Systems, Languages and Applications, Seattle, Washington, November 2002.

[BLS03] Chandrasekhar Boyapati, Barbara Liskov, and Liuba Shrira. Ownership Types for Object Encapsulation. Invited talk, Principles of Programming Languages, New Orleans, Louisiana, January 2003.

[Bok99] Boris Bokowski. Implementing "Object Ownership to Order." Proc. Intercontinental Workshop on Aliasing In Object-Oriented Systems, Lisbon, Portugal, June 1999.

[BN02] Anindya Banerjee and David A. Naumann. Representation Independence, Confinement, and Access Control. Proc. Principles of Programming Languages, Portland, Oregon, January 2002.

[BNR01] John Boyland, James Noble, and William Retert. Capabilities for Sharing: A Generalization of Uniqueness and Read-Only. Proc. European Conference on Object-Oriented Programming, Budapest, Hungary, June 2001.

[Boy01] John Boyland. Alias Burying: Unique Variables Without Destructive Reads. Software Practice & Experience, 6(31):533-553, May 2001.

[BR01] Chandrasekhar Boyapati and Martin Rinard. A Parameterized Type System for Race-Free Java Programs. Proc. Object-Oriented Programming Systems, Languages and Applications, Tampa, Florida, October 2001.

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

[Buc00] Alexander Buckley. Ownership Types Restrict Aliasing. MEng. Computing Final Year Project Report, Imperial College of Science, Technology and Medicine, London, United Kingdom, June 2000.

[BV99] Boris Bokowski and Jan Vitek. Confined Types. Proc. Object-Oriented Programming Systems, Languages, and Applications, Denver, Colorado, November 1999.

[CBS98] Edwin C. Chan, John T. Boyland, and William L. Scherlis. Promises: Limited Specifications for Analysis and Manipulation. Proc. International Conference on Software Engineering, Kyoto, Japan, April 1998.

[CD02] David Clarke and Sophia Drossopoulou. Ownership, Encapsulation, and the Disjointness of Type and Effect. Proc. Object-Oriented Programming Systems, Languages and Applications, Seattle, Washington, November 2002.

[CHP99] Karl Crary, Robert Harper, and Sidd Puri. What is a Recursive Module? Proc. Programming Language Design and Implementation, Atlanta, GA, June 1999.

[Cla01] David Clarke. Object Ownership & Containment. Ph.D. Thesis, University of New South Wales, Australia, July 2001.

[CNP01] David G. Clarke, James Noble, and John M. Potter. Simple Ownership Types for Object Containment. Proc. European Conference on Object-Oriented Programming, Budapest, Hungary, June 2001.

[CPN98] David G. Clarke, John M. Potter, and James Noble. Ownership Types for Flexible Alias Protection. Proc. Object-Oriented Programming Systems, Languages and Applications, Vancouver, Canada, October 1998.

[CRN03] Dave Clarke, Michael Richmond, and James Noble. Saving the World from Bad Beans: Deployment-time Confinement Checking. Proc. Object-Oriented Programming Systems, Languages and Applications, Anaheim, California, October 2003.

[CW03] Dave Clarke and Tobias Wrigstad. External Uniqueness is Unique Enough. Proc. European Conference on Object-Oriented Programming, Darmstadt, Germany, July 2003.

[CWM99] Karl Crary, David Walker, and Greg Morrisett. Typed Memory Management in a Calculus of Capabilities. Proc. Principles of Programming Languages, San Antonio, Texas, January 1999.

[DCH03] Derek Dreyer, Karl Crary, and Robert Harper. A Type System for Higher-Order Modules. Proc. Principles of Programming Languages, New Orleans, Louisiana, January 2003.

[DHT02] Eric M. Dashofy, André van der Hoek, Richard N. Taylor. An Infrastructure for the Rapid Development of XML-Based Architecture Description Languages. Proc. International Conference on Software Engineering, Orlando, Florida, May 2002.

[ELW98] Robert Eckstein, Marc Loy, and Dave Wood. Java Swing. O'Reilly & Associates, Sebastopol, California, September 1998.

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

[FD02] Manuel Fähndrich and Robert DeLine. Adoption and Focus: Practical Linear Types for Imperative Programming. Proc. Programming Language Design and Implementation, Berlin, Germany, June 2002.

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

[GAO94] David Garlan, Robert Allen, and John Ockerbloom. Exploiting Style in Architectural Design Environments. Proc. Foundations of Software Engineering, New Orleans, Louisiana, December 1994.

[GAO95] David Garlan, Robert Allen, and John Ockerbloom. Architectural Mismatch or Why it's Hard to Build Systems out of Existing Parts. Proc. International Conference on Software Engineering, Seattle, Washington, April 1995.

110

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

[GMJ+02] Dan Grossman, Greg Morrisett, Trevor Jim, Michael Hicks, Yanling Wang, and James Cheney. Region-Based Memory Mangagement in Cyclone. Proc. Programming Language Design and Implementation, Berlin, Germany, June 2002.

[GMW97] David Garlan, Robert T. Monroe, and David Wile. ACME: An Architecture Description Interchange Language. Proc. CASCON'97, Toronto, Ontario, November 1997.

[GPV01] Christian Grothoff, Jens Palsberg, and Jan Vitek. Encapsulating Objects with Confined Types. Proc. Object-Oriented Programming Languages, Systems, and Applications, Tampa, Florida, November 2001.

[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.

[Hog91] John Hogg. Islands: Aliasing Protection in Object-Oriented Languages. Proc. Object-Oriented Programming: Systems, Languages and Applications, Phoenix, Arizona, October 1991.

[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.

[ITU99] ITU-T. Recommendation Z.100, Specification and Description Language (SDL). Geneva, Switzerland, 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.

[KLR02] Viktor Kuncak, Patrick Lam, and Martin Rinard. Role Analysis. Proc. Principles of Programming Languages, Portland, Oregon, January 2002.

[LB85] M. M. Lehman, and Lazlo A. Belady. Program Evolution: Processes of Software Change. Academic Press, London, 1985.

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

[LLW03] Karl Lieberherr, David Lorenz, and Pengcheng Wu. A Case for Statically Executable Advice: Checking the Law of Demeter with AspectJ. Proc. Aspect-Oriented Software Development, Boston, Massachusetts, March 2003.

[LPZ02] K. Rustan M. Leino, Arnd Poetzsch-Heffter, and Yunhong Zhou. Using Ownership domains to Specify and Check Side Effects. Proc. Programming Language Design and Implementation, Berlin, Germany, June 2002.

[LR03] Patrick Lam and Martin Rinard. A Type System and Analysis for the Automatic Extraction and Enforcement of Design Information. Proc. European Conference on Object-Oriented Programming, Darmstadt, Germany, July 2003.

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

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

[Mad96] Testing Ada 95 Programs for Conformance to Rapide Architectures. Proc. Reliable Software Technologies - Ada Europe 96, Montreux, Switzerland, June 1996.

[Mic95] Microsoft Corporation. The Component Object Model Specification, Version 0.9. October 1995.

[Min96] Naftaly Minsky. Towards Alias-Free Pointers. Proc. of European Conference on Object Oriented Programming, Linz, Austria, July 1996.

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

[MNS01] Gail C. Murphy, David Notkin, and Kevin J. Sullivan. Software Reflexion Models: Bridging the Gap Between Design and Implementation. IEEE Trans. Software Engineering, 27(4), April 2001.

[Mon01] Robert Monroe. Capturing Software Architecture Design Expertise with Armani. Carnegie Mellon University technical report CMU-CS-98-163R, January 2001.

[MOR+96] Nenad Medvidovic, Peyman Oreizy, Jason E. Robbins, and Richard N. Taylor. Using Object-Oriented Typing to Support Architectural Design in the C2 Style. Proc. Foundations of Software Engineering, San Francisco, CA, October 1996.

[MP99] Peter Muller and Arnd Poetzsch-Heffter. Universes: A Type System for Controlling Representation Exposure. In A. Poetzsch-Heffter and J. Meyer (Hrsg.): Programmiersprachen und Grundlagen der Programmierung, 10. Kolloquium, Informatik Berichte 263, 1999/2000.

[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.

[MTH90] Robin Milner, Mads Tofte, and Robert Harper. The Definition of Standard ML. MIT Press, Cambridge, Massachussetts, 1990.

[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.

[Mye99] Andrew C. Myers. JFlow: Practical Most-Static Information Flow Control. Proc. Principles of Programming Languages, San Antonio, Texas, January 1999.

[NVP98] James Noble, Jan Vitek, and John Potter. Flexible alias protection. Proc. European Conference on Object-Oriented Programming, Brussels, Belgium, 1998.

[OCa00] Robert O'Callahan. Generalized Aliasing as a Basis for Program Analysis Tools. Ph.D. Thesis, published as Carnegie Mellon technical report CMU-CS-01-124, November 2000.

[OJ97] Robert O'Callahan and Daniel Jackson. Lackwit: A Program Understanding Tool Based on Type Inference. Proc. International Conference on Software Engineering, Boston, Massachusetts, May 1997.

[OMG95] Object Management Group. The Common Object Request Broker: Architecture and Specification (CORBA), revision 2.0. 1995.

[Par72] David L. Parnas. On the Criteria to be Used in Decomposing Systems into Modules. Communications of the ACM, 15(12):1053--1058, December 1972.

[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.

[Rey02] John C. Reynolds. Separation Logic: a Logic for Shared Mutable Data Structures. Proc. Logic in Computer Science, Copenhagen, Denmark, July 2002.

[RFS+00] Alastair Reid, Matthew Flatt, Leigh Stoller, Jay Lepreau, and Eric Eide. Knit: Component Composition for Systems Software. Proc. Operating Systems Design and Implementation, San Diego, California, October 2000.

112

[Ric02] John David Richmond. Report on a Case Study Applying ArchJava to the DynamicJava Interpreter. Unpublished manuscript, June 2002.

[RJB98] James Rumbaugh, Ivar Jacobson, and Grady Booch. The Unified Modeling Language Reference Manual. Addison-Wesley, 1998.

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

[RSC00] Rational Software Corporation. Rational Rose RealTime. http://www.rational.com/, 2000.

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

[SD03] Matthew Smith and Sophia Drossopoulou. Cheaper Reasoning with Ownership Types. Proc. International Workshop on Aliasing, Confinement, and Ownership in Object-Oriented Programming, Darmstadt, Germany, July 2003.

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

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

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

[Sun00]   Sun Microsystems, Inc.   Enterprise Java Beans Specification.   Available at http://java.sun.com/ejb, April 2000.

[SY02] Mati Shomrat and Amiram Yehudai. Obvious or Not? Regulating Architectural Decisions using Aspect-Oriented Programming. Proc. Aspect-Oriented Software Development, Enschede, The Netherlands, April 2002.

[TT94] Mads Tofte and Jean-Pierre Talpin. Implementing the Call-by-Value λ-Calculus Using a Stack of Regions. Proc. Principles of Programming Languages, Portland, Oregon, January 1994.

[Wad90a] Philip Wadler. Linear Types Can Change the World! Programming Concepts and Methods, (M. Broy and C. Jones, eds.) North Holland, Amsterdam, April 1990.

[Wad90b] Philip Wadler. Comprehending Monads. Proc. Lisp and Functional Programming, Nice, France, June 1990.

[WM00] David Walker and Greg Morrisett. Alias Types for Recursive Data Structures. Proc. International Workshop on Types in Compilation, Montreal, Canada, September 2000.

Vita

Research Interests

Programming languages, software engineering, compilers, and parallel and distributed systems.

Education

Ph.D., Computer Science, University of Washington, August 2003.

    Advisors: Craig Chambers and David Notkin

    Thesis: ArchJava: Connecting Software Architecture to Implementation

M.S., Computer Science, University of Washington, June 1999.

B.S., Engineering and Applied Science (Computer Science), California Institute of Technology, June 1997.

Teaching Experience

Winter 2002. Teaching Assistant, CSE 503 (Graduate Software Engineering), University of Washington.

Winter 2001. Teaching Assistant, CSE 501 (Graduate Compilers), University of Washington.

Summer 1999. Pre-Doctoral Lecturer, CSE 143 (Computer Programming II), University of Washington.

Spring 1999. Teaching Assistant, CSE 143 (Computer Programming II), University of Washington.

Fall 1998. Teaching Assistant, CSE 505 (Graduate Programming Languages). University of Washington.

Professional Experience

2003-present. Assistant Professor, Carnegie Mellon University.

1997-2003. Graduate Student and Research Assistant, University of Washington.

2000-2003. Developed ArchJava, a small extension to Java that integrates a software architecture specification into Java source code. ArchJava is the first general-purpose object-oriented language that verifies *communication integrity*: all run-time communication due to control flow and shared data is explicitly declared in the architecture. Case studies suggest that ArchJava can be easily applied to existing Java programs, and that it may provide benefits when developing and evolving code.

1998-2000. Developed, implemented, and evaluated static analyses to remove unnecessary synchronization from Java programs. The analyses remove nearly all unnecessary synchronization on most benchmarks, and improved performance by 37-50% in synchronization-intensive applications.

1997-1998. Implemented thread support in the runtime system of the Vortex optimizing compiler.

Summer 1997. Research Assistant, California Institute of Technology.

Implemented a distributed object system for Java that provides location transparency, persistence, agent mobility, and fault-tolerant communication.

Summer 1996. Intern, Sequent Computer Systems, Inc.

> Developed web-based technologies for a Corporate Digital Library information management system.

Summer 1995. Intern, Sequent Computer Systems, Inc.

> Implemented a simulator for a massively parallel video-on-demand server. Developed a novel communication optimization that increased simulated effective video bandwidth by 50%.

Summer 1994. Intern, Sequent Computer Systems, Inc.

> As part of a team, converted an in-house decision support technology into a product that was later spun off into a successful company, DecisionPoint Applications.

Summer 1993. Intern, Sequent Computer Systems, Inc.

> Benchmarked two object-oriented databases in C++ and Java running on a parallel Sequent server to evaluate their scalability.

Summer 1992. Intern, Adaptive Technology Applications, Inc.

> Used neural-network technology to recognize the hippocampus in MRI scans of patient's brains as a possible diagnostic tool for Alzheimer's disease.

Selected Honors

1997-2000 National Defense Science and Engineering Graduate Fellowship

1997-2000 Achievement Rewards for College Scientists Fellowship

1997 National Science Foundation Fellowship Honorable Mention

1995-1997 Caltech Merit Scholarship

1993 National Merit Scholarship

1996 Winner, Caltech-Occidental Symphony Concerto Competition (violin)

Honor societies: Sigma Xi (scientific research), Tau Beta Pi (engineering)

Refereed Journal and Conference Publications

Jonathan Aldrich, Vibha Sazawal, Craig Chambers, and David Notkin. Language Support for Connector Abstractions. In Proceedings of the *European Conference on Object-Oriented Programming (ECOOP '03)*, July 2003.

Jonathan Aldrich, Valentin Kostadinov, and Craig Chambers. Alias Annotations for Program Understanding. In Proceedings of *Object-Oriented Programming Systems, Languages, and Applications (OOPSLA '02)*, November 2002.

Jonathan Aldrich, Craig Chambers, and David Notkin. Architectural Reasoning in ArchJava. In Proceedings of the *European Conference on Object-Oriented Programming (ECOOP '02)*, June 2002.

Jonathan Aldrich, Craig Chambers, and David Notkin. ArchJava: Connecting Software Architecture to Implementation. In Proceedings of the *International Conference on Software Engineering (ICSE '02)*, May 2002.

Jonathan Aldrich, Emin Gun Sirer, Craig Chambers, and Susan Eggers. Comprehensive Synchronization Elimination for Java. To appear in *Science of Computer Programming*.

Jonathan Aldrich, Craig Chambers, Emin Gun Sirer, and Susan Eggers. Static Analyses for Eliminating Unnecessary Synchronization from Java Programs. In Proceedings of the *Sixth International Static Analysis Symposium (SAS '99)*, September 1999.

Workshop Papers and Technical Reports

Jonathan Aldrich, Vibha Sazawal, Craig Chambers, and David Notkin. Architecture-Centric Programming for Adaptive Systems. In Proceedings of the *Workshop on Self-Healing Systems (WOSS '02)*, November 2002.

Vibha Sazawal and Jonathan Aldrich. Architecture-Centric Programming for Context-Aware Configuration. In Proceedings of the *OOPSLA '02 Workshop on Engineering Context-Aware Object-Oriented Systems and Environments (ECOOSE '02)*, November 2002.

Jonathan Aldrich. Challenge Problems for Separation of Concerns. In Proceedings of the *OOPSLA 2000 Workshop on Advanced Separation of Concerns*, October 2000.

Jonathan Aldrich. Evaluating Module Systems for Crosscutting Concerns. *University of Washington Ph.D. Generals Examination Report*, September 2000.

Jonathan Aldrich, James Dooley, Scott Mandelsohn, and Adam Rifkin. Providing Easier Access to Remote Objects in Client Server Systems. In *Thirty-first Hawaii International Conference on System Sciences (HICSS-31)*, January 1998.

Service

Program Committee: ECOOP '03 Workshop on Ownership, Confinement, and Aliasing

Reviewer: OOPSLA '02, POPL '03, The Internet Encyclopedia

University of Washington Department of Computer Science & Engineering:

>   2002 Co-organized graduating students support group
>
>   2002 Graduate Admissions Committee
>
>   2001-2002 Faculty Recruitment Liaison
>
>   1998-2000 Prospective Graduate Student Recruiting Committee
>
>   1998-2000 Volunteer Tutor

Green Lake Church: 1999-2002 church board, 1998-2001 music committee

Caltech Christian Fellowship: 1995-1996 Social Director; 1996-1997 President