Using Types to Enforce Architectural Structure

Jonathan Aldrich
School of Computer Science
Carnegie Mellon University
5000 Forbes Ave, Pittsburgh, PA 15213
jonathan.aldrich@cs.cmu.edu

Abstract

Developers use formal or informal software architecture descriptions in order to communicate and reason about the high-level structural properties of a system. However, these architectural descriptions are often inaccurate or out of date, resulting in lost productivity and defects as the system is evolved.

This paper presents the first static technique for enforcing complete structural conformance between a rich architectural specification and general-purpose implementation code. Our system, ArchJava, models architecture as a hierarchy of component instances that communicate through explicit connections. ArchJava's type system ensures that components only communicate through connections that are explicitly declared in the architecture. As a result, developers have accurate architectural documentation, allowing them to carry out evolution tasks with confidence.

To validate our design, we show how ArchJava can be used to capture the Mission Data System architecture under development at JPL for embedded space system applications.

1. Introduction

Software architecture is the high-level organization of a software system, showing how the system decomposes into components, and how these components interact [GS93,PW92]. Various systems have been developed to allow architects to specify and reason about different architectural properties, including the temporal order of architectural events [LV95,AG97], architectural styles [AAG93,MOR+96], and the evolution of dynamic systems [MK96]. One major, long-term goal of software architecture research is to aid engineers in development and evolution tasks by enforcing these architectural properties in the implementation of a system.

All of the properties cited above rely on a basic notion of architectural structure: a description of how the major components in a software system interact. For example, modeling the temporal order of architectural events includes specifying where these events occur in the architecture; architectural styles constrain the topology of an architecture and how components can communicate; and architectural dynamism involves structure that changes over time. Thus, a necessary first step towards enforcing any of these properties in a real system is enforcing structural conformance between architecture and implementation code.

A system conforms to its architecture if the architecture is a conservative abstraction of the run-time behavior of the system. The *communication integrity* property defines how architectural structure abstracts 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 which support implicit communication, including references, objects, and first-class functions. Previous systems have made serious compromises in order to enforce communication integrity, either eliminating implicit communication mechanisms entirely [ITU99], postponing conformance checks until run time [Mad96], or supporting only simple architectural models [MNS01,LR03].

We previously presented the initial design of ArchJava, which allows programmers to model rich architectural designs within Java code [ACN02a]. ArchJava allows developers to label distinguished objects as architectural components, and specify how those components interact through connections. The previous version of ArchJava enforced communication integrity for function calls between components, but did not enforce integrity for communication through shared data.

This paper makes the following technical contributions:

- We show how ArchJava can be extended to describe architectural constraints on data sharing by adapting our previous work on alias control systems [AKC02,AC04]. Our system can describe data that is confined within a component, passed linearly from one component to another, or shared temporarily or persistently between components.
- The extended ArchJava design is the first system to statically enforce communication integrity for rich architectural models in the presence of data sharing. We define communication integrity precisely for ArchJava and explain how the checking is done.

- We validate the extended ArchJava design in practice, showing how it can be used to capture the Mission Data System architecture under development at JPL for embedded space system applications.

In the next section, we review the alias control constructs of AliasJava, the alias-control type system on which we build. Section 3 shows how these constructs can be integrated into ArchJava to support a specification of data sharing in an architecture. Section 4 defines communication integrity precisely for ArchJava, and explains how it is checked. Section 5 shows how ArchJava can be used to capture the architecture of JPL's MDS architecture. Section 6 discusses related work, and Section 7 concludes.

2. AliasJava

AliasJava is a type annotation system that extends Java to express how data is confined within, passed among, or shared between components and objects in a software system [AKC02,AC04]. The ArchJava language, discussed in Section 3, builds on this foundation by adding constructs for describing software architecture.

2.1. Alias-Control Model

The goal of AliasJava is to enforce a high-level specifications of aliasing relationships in object-oriented programs. We achieve this goal by dividing objects into conceptual groups called ownership domains, and allowing architects to specify high-level policies that govern references between ownership domains. Ownership domains are hierarchical, allowing engineers to specify very abstract aliasing constraints at the level of an entire program, then refine these constraints to specify aliasing within subsystems, modules, and individual objects.

AliasJava supports abstract reasoning about data sharing by assigning each object in the system to a single ownership domain. There is a top-level ownership domain denoted by the keyword **`shared`**. In addition, each object can declare one or more domains to hold its internal objects, supporting hierarchical aliasing specifications.

For example, Figure 1 uses a Sequence abstract data type to illustrate the ownership model used in AliasJava. The Sequence object and its clients are both part of the top-level *shared* ownership domain. Within the sequence, the *iters* ownership domain is used to hold iterator objects that clients use to traverse the sequence, and the *list* ownership domain is used to hold the cons cells in the linked list that is used to represent the sequence.

Each object can declare a policy describing the permitted aliasing among objects in its internal domains, and between its internal domains and external domains. AliasJava supports two kinds of policy specifications:

- A *link* from one domain to another, denoted with an dashed arrow in the diagram, allows objects in the first domain to access objects in the second domain.

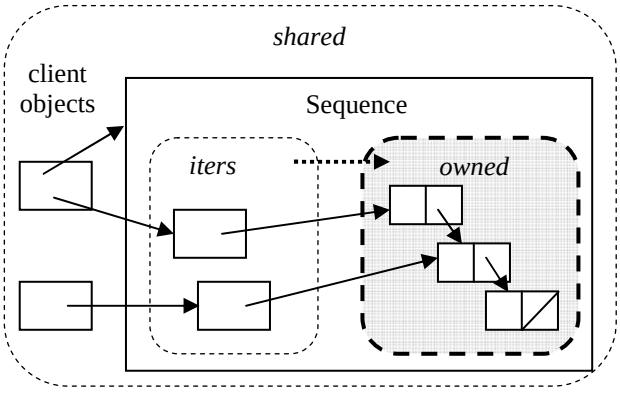


Figure 1. A conceptual view of the aliasing model used in AliasJava and ArchJava. The rounded, dashed rectangles represent ownership domains, with a gray fill for private domains. Solid rectangles represent objects. The top-level *shared* domain contains the highest-level objects in the program. Each object may define one or more domains that in turn contain other objects.

- A domain can be declared *public*, denoted by a thinner dashed rectangle with no shading. Permission to access an object automatically implies permission to access its public domains.

For example, in Figure 1 the Sequence object declares a link from its *iters* domain to its *owned* domain, allowing the iterators to refer to objects in the linked list. The *iters* domain is public, allowing clients to access the iterators, but the *owned* domain is private, and so clients must access the elements of the sequence through the iterator interface rather than traversing the linked list directly.

In addition to the explicit policy specifications mentioned above, our system includes the following implicit policy specifications:

- An object has permission to access other objects in the same domain.
- An object has permission to access objects in the domains that it declares.

The first rule allows the clients to access the sequence (and vice versa), while the second rule allows the sequence to access its iterators and linked list. Any references not explicitly permitted by one of these rules is prohibited, according to the principle of least privilege. It is crucial to this example that there is no transitive access rule: for example, even though clients can refer to iterators and iterators can refer to the linked list, clients cannot access the linked list directly because the sequence has not given them permission to access the owned domain. Thus, the policy specifications allow developers to specify that some objects are an internal part of an abstract data type's representation, and the compiler enforces the policy, ensuring that this representation is not exposed.

```
class Sequence<T> {
  domain owned; /* default */
  owned Cons<T> head;
  void add(T o) {
    head = new Cons<T>(o,head)
  }

  public domain iters;
  link iters -> owned;
  iters Iterator<T> getIter() {
    return new SequenceIterator<T, owned>(head);
  }
}

class Cons<T> {
  T obj;
  owner Cons<T> next;

  Cons(T obj, owner Cons<T> next) {
    this.obj=obj; this.next=next; }
}
```

Figure 2. A `Sequence` abstract data type that uses a linked list for its internal representation. The `Sequence` declares a publicly accessible *iters* domain representing its iterators, as well as a private *owned* domain to hold the linked list. The link declarations specify that iterators in the *iter* domain have permission to access objects in the *owned* domain, and that both domains can access owner of the type parameter `T`.

2.2. Alias Annotations.

Figure 2 shows how the Java code defining the sequence ADT can be annotated with aliasing information to model the constraints expressed in Figure 1. The `Sequence` class is parameterized by the type `T` of its element objects, using Java version 1.5's generics support.

The first two lines of code within the class declare the *owned* domain and a reference to the head of the list. For convenience, every object in our system declares its own *owned* domain, and so we will omit this declaration from future examples. The `head` field is of type `owned Cons<T>`, denoting a `Cons` linked list cell that holds an element of type `T` and resides in the `owned` domain. The `add` member function constructs a new cons cell for the object passed in, and adds it to the head of the list.

Skipping ahead to the definition of the `Cons` cell class, we see that it is also parameterized by the element type `T`. The class contains a field `obj` holding an element in the list, along with a `next` field referring to the next cons cell (or **`null`**, if this is the end of the list). The `next` field has type `owner Cons<T>`, indicating that the next cell in the list has the same owner domain as the current cell (i.e., all the cells are part of the `Sequece`'s `owned` domain).

Back in the `Sequence` class, a public *iters* domain is declared to hold the iterator objects. Because the iterators need to refer to cons cells in the linked list, the sequence links the *iter* domain to the *owned* domain. The `getIter` method creates a `SequenceIterator` object (not shown), initializing the iterator to point to the first element of the linked list.

Uniqueness and Lending. While ownership is a useful for representing persistent aliasing relationships, it cannot capture the common scenario of an object that is passed between objects without creating persistent aliases. Objects to which there is only one reference (including newly-created objects) are annotated **`unique`** in AliasJava. Unique objects can be passed from one ownership domain to another, as long as the reference to the object in the old ownership domain is destroyed when the new reference is created.

We also allow one ownership domain to temporarily lend an object to another ownership domain, with the constraint that the second ownership domain will only use the object in the course of a particular function call and will not create any persistent references to the object. We annotate these temporary references with the keyword **`lent`**, and enforce the invariant that **`lent`** references cannot be stored in object fields.

2.3. Properties.

AliasJava enforces a *policy soundness* property, ensuring that the aliasing policy specifications in the program text are obeyed at run time:

> **Definition [Policy Soundness]:** If in object that is part of ownership domain D_1 refers to an object in domain D_2, then there must be a policy specification allowing references from D_1 to D_2.

Policy soundness is crucial to enforcing communication integrity in the presence of data sharing, as described below, because it ensures that the data sharing declarations in a software architecture are obeyed at run time.

Policy soundness is enforced statically by AliasJava's type system, by ensuring consistency among ownership annotations and by making sure references between objects are legal given the policy specifications in scope. Our previous paper proved a policy soundness property in a formal model of the AliasJava language [AC04].

Summary. AliasJava uses type annotations to partition an object's internal state into disjoint ownership domains. Policy specifications constrain inter-domain aliasing, so that objects in one domain can only refer to objects in another domain if the policy allows these references. In the next section, we show how ArchJava leverages AliasJava's ownership domains in architectural specifications to control communication through shared data.

3. ArchJava

ArchJava extends the Java language with component classes, which describe objects that are part of an architecture, connections, which allow components to communicate, and ports, which are the endpoints of connections. Components are organized into a hierarchy using owner-


```
public component class GraphicsPipeline {
  protected owned Generate generate = ... ;
  protected owned Transform transform = ... ;
  protected owned Rasterize rasterize = ... ;

  connect pattern Generate.out, Transform.in;
  connect pattern Transform.out, Rasterize.in;

  public GraphicsPipeline() {
    connect(generate.out, transform.in);
    connect(transform.out, rasterize.in);
  }
}

public component class Transform {
  protected owned Trans3D currentTransform;

  public port in {
    provides void draw(unique Shape s);
  }
  public port out {
    requires void draw(unique Shape s);
  }

  void draw(unique Shape s) {
    currentTransform.apply(s);
    out.draw(s);
  }
}
```

Figure 3. The architectural specification of a graphics pipeline in ArchJava. `GraphicsPipeline` is made up of three subcomponents: the `Generator` generates shapes, which are transformed by `Transform` and then displayed by `Rasterize`. The `Transform` component accepts a `unique Shape` on its `in` port, transforms it according to the current transformation, and passes it on through the `out` port.

ship domains, and ownership domains can be shared along connections, permitting the connected components to communicate through shared data. This section introduces these concepts through two example architectures.

3.1. Example: Pipeline Architecture

Figure 3 shows the architecture of a simple graphics pipeline. The `generate` component stores the current scene and generates shapes to be displayed. These shapes are passed on to the `transform` component, which stores the current transformation and applies it to each shape in turn. It then passes the shapes on to the `rasterize` component to be displayed.

We want to enforce two architectural invariants that are important to the pipeline architectural style [GS93]. First, the components are arranged in a linear sequence, with each component getting information from its predecessor and sending it on to its successor. Second, no data is shared between components; instead, shapes are handed off from one component to another. As the ArchJava language is introduced through this example, we will discuss how these invariants are specified and enforced.

3.2. Components and the Ownership Hierarchy

A *component* in ArchJava is a special kind of object whose communication patterns are declared explicitly using architectural declarations. Component code is defined in ArchJava using *component classes*. Figure 3 shows the code that defines the `GraphicsPipeline` and `Transform` component classes. We assume that `Generate` and `Rasterize` are component classes defined elsewhere, and `Trans3D` and `Shape` are ordinary classes that are not part of the architecture.

The `GraphicsPipeline` class contains three fields, one for each component in the pipeline. The fields types are annotated with the implicit ownership domain **`owned`**, meaning that `generate`, `transform`, and `rasterize` are *subcomponents* of the `GraphicsPipeline` component instance that owns them.

3.3. Ports and Unique Data

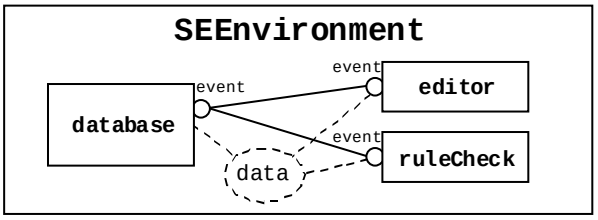
Components communicate through explicitly declared ports. A *port* is a communication endpoint declared by a component. For example, the `Transform` component class declares an `in` port that receives incoming shapes and an `out` port that passes transformed shapes on to the next component.

Each port declares a set of required and provided methods. 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. Each provided method must be implemented inside the component. For example, the `draw` method's implementation transforms its shape argument and then calls the required method `draw` on the `out` port. As the example shows, a component can invoke one of its required methods by sending a message to the port that defines the required method.

Annotating the `Shape` objects as **`unique`** enforces the architectural invariant that shapes are handed off from one component to another. ArchJava's type system ensures that no component may retain a reference to a shape after it passes it on to the next component. This invariant allows the developers of each component to assume they have exclusive access to the shape they are manipulating.

3.4. Connections and Connect Patterns

ArchJava requires developers to declare in the architecture the connection patterns that are permitted at run time. The declaration **`connect pattern`** `Generate.out, Transform.in` permits the graphics pipeline component to make connections between the `out` port of its `Generate` subcomponents and the `in` port of its `Transform` subcomponents. The connect patterns declared in `GraphicsPipeline` constrain its subcom-




```
public component class SEEnvironment {
  protected owned Database database = ... ;

  connect pattern Database.event, Tool.event;

  public void instantiateTool(Class tCls) {
    owned Tool tool = (Tool)tCls.newInstance();
    connect(database.event, tool.event);
    tool.initialize();
  }

  // reads config file, calls instantiateTool...
}

public abstract component class Tool {
  public port event {
    domain data;
    requires void signal(unique Event e);
    requires void register(unique EventType t,
                           data Callback cb);
  }
}
```

Figure 4. The architectural specification of a software engineering environment. The environment is made up of a central database that stores the code for the project, and a set of tools that communicate through events that are mediated by the database.

ponents to communicate in a linear sequence, fulfilling the constraint of the pipeline architectural style.

Once connect patterns have been declared, concrete connections can be made between components. All connected components must be part of an ownership domain declared by the component making the connection. For example, the constructor for `GraphicsPipeline` connects the `out` port of the `transform` component instance to the `in` port of the `rasterize` component instance. This connection binds the required method `draw` in the `out` port of `transform` to a provided method with the same name and signature in the `in` port of `rasterize` (not shown). Thus, when `transform` invokes `draw` on its `out` port, the corresponding implementation in `rasterize` will be invoked.

3.5. Example: Blackboard Architecture

Figure 4 shows the architecture of a software engineering environment. The architecture is structured as a blackboard, with various tools accessing a central database that stores the code base on which the tools operate [GS93]. In the architectural diagram, the oval represents an ownership domain holding the data that is shared between the database and all the components. The architectural invariant of the system is that tools communicate only through the shared data and via events that are mediated by the central database [SN92].

The `SEEnvironment` component class declares the code database as an **`owned`** component. However, it doesn't declare a fixed set of components at the architectural level, because we would like the environment to be extensible, loading tools at run time that may have been developed by third parties. Therefore, the architecture declares a connect pattern between the `event` port of the database and the `event` port of the abstract component class `Tools`.

`SEEnvironment` reads a configuration file to determine the set of installed components and then instantiates them one by one using the `instantiateTool` function. This function takes a component class argument, creates a new component instance, and casts the instance to type `Tool`. The tool is then connected to the database using a connect expression that matches the connect pattern in the architecture, and finally the tool is initialized. This design allows an arbitrary number of tools to be created and linked into the software engineering environment.

Shared ownership domains. Components can share objects with connected components by declaring ownership domains inside their ports. When the port is connected to a matching port, ownership domains with the same name that are declared in both ports are merged, allowing both components to access the objects in the shared domain.

The `event` port in component class `Tool` shows how the tools communicate with the database. The `data` ownership domain describes the objects that are shared between the database and all the tools, including the code stored in the database and callback objects that react to events.

Every tool can signal an event by invoking the `signal` function. The event passed to `signal` is **`unique`**; it will be enqueued in the database event queue before being delivered to tools that have expressed interest in events of that type.

Tools can also register for events of a particular type by passing in a **`unique`** event descriptor object, together with a callback that will be invoked when an event occurs. The callback is expected to define a `notify` method that will be invoked with the event argument.

The `event` port of `Database` (not shown) is the mirror of the `event` port of `Tool`. It also declares the `data` domain and defines provided methods `signal` and `register` that match the methods declared in the port of `Tool`.

An Example Tool. The `RuleChk` component in Figure 5 is intended to ensure that the code base obeys a set of user-defined coding rules. It stores the set of rules in some internal format in the `ruleSet` object. When initialized, it registers a callback to be invoked whenever any change to the code occurs.

```
public component class RuleChk extends Tool {
  protected owned Set<owned> ruleSet;

  public port event {
    domain data;
    requires void signal(unique Event e);
    requires void register(unique EventType t,
                           data Callback cb);
  }

  public void initialize() {
    event.register(new EventType("codeChange"),
                   new RuleCB<owned>(ruleSet));
  }
}

class RuleCB<rules> implements Callback {
  protected rules Set<rules> ruleSet;

  RuleCB(rules Set<rules> rs) { ruleSet = rs; }

  void notify(lent Event e) {
    // generates an error on rule violations
  }
}
```

Figure 5. The `RuleChk` component stores a set of semantic rules, and registers a callback to receive code change events. Whenever the callback is invoked with an event, it checks if any of the rules are violated, and if so it generates an error.

The callback object needs to access the set of rules, so the class is parameterized by the domain that holds the rules, which is instantiated with the **`owned`** domain of `RuleChk`. It stores the `ruleSet` internally in a field annotated with this domain.

When a code change event is fired, the `notify` method of the `RuleCB` callback will be invoked. We assume that the database owns the events in the system, but callback objects need to have temporary access to the event object in order to get information about the event. Therefore, the database passes the event to the callback as a **`lent`** reference. The callback checks to see if the the event leads to a rule violation, and notifies the user if a violation is detected.

This example illustrates ArchJava's support for event callback objects, and important object-oriented idiom that is challenging to reason about in conventional implementation languages.

3.6. Implementation

An open-source compiler for ArchJava is available for download at the ArchJava web site [Arc02]. Our compiler is implemented on top of the Barat infrastructure [BS98]. The compiler accepts a list of ArchJava files (.archj), translates each one down into Java source code, and invokes `javac` on the resulting .java files. Both typechecking and compilation are local, so that when a source file is updated, only that file and the files that depend on its interface need to be typechecked and recompiled.

The most interesting aspect of compiling ArchJava is that some information about ownership domains must be maintained at run time, using standard type-passing techniques. Although ArchJava's type system is mostly static, ArchJava performs run-time checks at downcasts and array writes to ensure that the domain parameters of an object match the parameters declared in the type of the cast or array. These checks are done at the same places where Java already does dynamic checks; in this sense, ArchJava's type system is as static as that of Java. Other papers provide additional details about the type system and the implementation techniques used in the compiler [ACN02b,AKC02,Ald03].

3.7. 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 communication through method calls and shared data.

4. Communication Integrity

Communication integrity is the key property enforced by ArchJava, ensuring that components can only communicate using connections and ownership domains that are explicitly declared in the architecture. In this section, we define communication integrity more precisely, justify the definition, and explain how it is enforced.

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`**, it has no architectural domain.

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

1. **Direct call:** Component instance `A` or an object in one of its ownership domains invokes a method directly on 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* (invokes a method or reads or writes a field of) a non-component object `B`, and `A` and `B` are in different architectural domains.

We now state the communication integrity theorem for ArchJava:

Theorem [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 B that is annotated **`unique`**, or
2. **Parent-child communication:** Object A invokes a method on a component B which is owned by A, or
3. **Connection communication:** Component A invokes a method on component B through a connection that matches a connect pattern in the component instance that directly owns (or is equal to) A and B, or
4. **Lent communication:** Component or object A invokes a method on an object or component B that has been temporarily lent to A, or
5. **Shared domain communication:** Object A accesses some object B in a different domain, and the architectural domain of A is linked to that of B.

The author's thesis includes a formal model of the ArchJava language, a formal statement of the communication integrity theorem described above, and a rigorous proof that ArchJava's type system statically enforces communication integrity [Ald03]. Below, we outline the structure of the proof and provide an intuition for how the property is enforced.

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.** ArchJava's type system ensures if the receiver of a method call is a component, then either the receiver is **`this`**, or the receiver is **`unique`** or part of a locally declared ownership domain. In the case of **`this`**, the communication is within a component. In the cases of **`unique`** and local domains, the communication is unique communication and parent-child communication, respectively.
2. **Connection call case.** The type system must ensure that the component which owns both the sender and the receiver declared a connection between them. When 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 **`owned`**, again, there is no inter-component communication, because the receiver of the access is part of the same component as the sender. If the annotation is a lent domain parameter, the communication is lent communication.

   The remaining case is when the accessed object is annotated with a ownership domain that is either declared in the current component. We wish to show that this case is shared domain communication. This will be true if and only if architectural domain of the accessing object can access the target object's domain according to the aliasing policy. But this is guaranteed by the policy soundness property, so we are done.

Discussion. The theoretical framework described above is quite general—for example, communication through static fields or native methods can be modeled as shared domain communication, where the fields and native methods are conceptually viewed as part of the *shared* domain that is shared between every component. In practice, however, excessive communication through the global *shared* domain makes architectural reasoning more difficult, and so developers are encouraged to avoid it where possible, just as good engineers typically avoid using global variables in today's programming languages. We would prefer to omit the global *shared* domain entirely, but this would be impractical given that many existing Java libraries use global data structures.

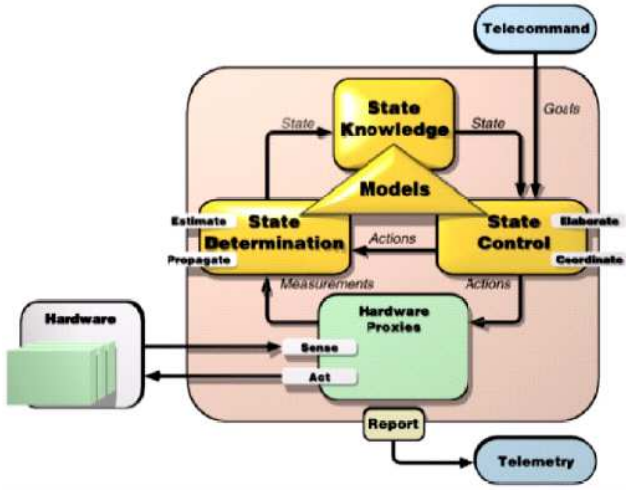
Communication integrity means that all communication between components must be declared at the architectural level—either through required and provided methods in connected ports, or through an ownership domain declared in connected ports. The ArchJava compiler enforces conformance via local rules governing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

5. Validation

In order to validate our design, we are undertaking a case study applying ArchJava to Golden Gate, the real-time robot control system developed by the NASA Jet Propulsion Laboratory, Sun, and Carnegie Mellon University, and demonstrated at the JavaOne conference in 2003. The goal of the study is to answer the following research questions:

- Can ArchJava effectively capture the Mission Data System architecture used in the Golden Gate code?
- What are the benefits of ArchJava, in terms of understanding the actual architecture of the code, and in finding possible violations of the intended architecture?
- What are the costs of using ArchJava in this system?

While our case study is not yet complete, we report on our preliminary experience with this application.




```
public component class ControlDiamond {
  protected owned State state;
  protected owned Estimator estimator;
  protected owned Control control;
  protected owned Hardware hardware;

  public port telemetry { ... }
  public port report { ... }

  cnct pat Estimator.estimate, State.data;
  cnct pat State.data, Control.state;
  cnct pat Hardware.measure, Estimator.measure;
  cnct pat Control.action, Hardware.action,
    Estimator.action;

  glue telecommand to control.goals;
  // additional code not shown
}
```

Figure 6. A graphical depiction of the Mission Data System architecture in use at the Jet Propulsion Laboratory, and simplified ArchJava code that captures the architecture.

Architecture. The core of the Mission Data System (MDS) architecture [DRR+99] used by Golden Gate is shown graphically at the top of Figure 6. The architecture shown is designed to capture the state estimation and control loop for a single element of hardware within the robot. A Hardware Proxy, at the bottom, communicates directly with the hardware, reporting measurements to the rest of the system and accepting action commands to be performed by the hardware. To the left is a State Determination (or estimator) component that takes measurements from the hardware proxy and uses them to estimate the value of some higher-level state, such as the robot's current position. This state is then stored in the State Knowledge component at the top of the diagram. Finally, the State Control component on the right accepts external commands from mission control through an external telemetry module (at the top-right) and uses information about the current state to determine what actions to perform next. An application's complete architecture is made of a number of these diamond-shaped subarchitectures.

We chose to represent this with the `ControlDiamond` component class shown at the bottom of Figure 6.[1] The four subcomponents in the diagram are represented by four **`owned`** components. The `ControlDiamond` has two external ports, one for receiving commands from mission control and one for reporting information back to mission control. The connection to hardware in the diagram is not represented in ArchJava, as this is done through native methods that are beyond the scope of our language design.

Connections in the architecture correspond to the arrows in the diagram. The last connection uses the keyword **`glue`**, indicating that goals coming from the telemetry port should be forwarded directly to the `goals` port of the `control` subcomponent, as shown in the diagram.

Discussion. The original Golden Gate code was designed with the MDS architecture in mind, and so the source code refers explicitly to concepts like components and connectors, making our task easier. On the other hand, it was still somewhat challenging to associate code with architectural features, because connections were made by calls to two *connect* functions deep within the (quite complex) constructor code of the `ControlDiamond`. Furthermore, one of the two *connect* functions does not describe what interface is used for communication; this is inferred from the types of the two connected components. As Figure 6 shows, our ArchJava representation provides a clearer view of the architecture by declaring architectural connections at the top level, and using ports to show the interfaces between components.

One difference we found between the abstract architecture and the code was that often the state in one "diamond" in the architecture is used by the control or estimation components in another diamond. This is a natural requirement of the domain, where different state variables are somewhat interdependent, but this was not explicit in the original architectural diagram.

It is too early in our case study to evaluate the costs of applying ArchJava; our previous experience was that ArchJava can be applied to 10K-line legacy Java systems in about 6-30 engineer hours [ACN02a,ACN02b].

6. Related Work

ArchJava. The initial ArchJava system enforced architectural conformance only for control flow between components, not for communication though shared data [ACN02a, ACN02b]. This paper extends our previous work to the more challenging case of communication through shared data, enforcing communication integrity for all forms of communication.

In addition, the system we describe here is more flexible and more consistent than our previous system. For example, the component hierarchy is specified using own-

[1]Note: because of export restrictions the code shown is not actual Golden Gate code, but rather is an abstracted view demonstrating how we are capturing the architecture.

ership domains, rather than the ad-hoc and inflexible syntactic criterion used before. One benefit is that we can now support the factory pattern [GHJ+94] for components: a factory component creates and initializes components, which are then passed as a **`unique`** component to their final place in the architecture, where they become **`owned`** by their parent component. Another benefit is that Java constructs like inner classes, interface inheritance, and native methods fit more cleanly into our current framework, as discussed elsewhere [Ald03].

We believe these improvements make ArchJava considerably more practical, and that the support for full architectural conformance will provide significant benefits to users of the language.

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

Module Systems. Module systems such as ML's functors [MTH90] and MzScheme's Units [FF98] support system composition from separate modules. While these module systems have rich facilities for information hiding, they do not provide mechanisms for controlling shared data objects or functions, and thus do not enforce architectural conformance.

Enforcing Design. 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. Furthermore, their system does not describe data sharing as precisely, omitting constructs like uniqueness and ownership-based encapsulation. However, they do describe a number of useful analyses which would complement ArchJava's more detailed architectural descriptions.

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]. This analysis-based approach is more lightweight than ArchJava's type system, but does not support hierarchical, dynamic architectures or precise data sharing constraints.

CASE Tools. Several CASE tools support the SDL language, which allows developers to describe architectural structure within the implementation of an embedded system [ITU99]. The language enforces architectural conformance, but only by prohibiting shared references between components. Other CASE tools such as Rational Rose RealTime [RSC00] also allow developers to specify the design of a system, but in the presence of shared objects and references they do not enforce architectural conformance.

Ownership and Uniqueness. Ownership was introduced in the Flexible Alias Protection paper, which uses ownership polymorphism to strike a balance between guaranteeing aliasing properties and allowing flexible programming idioms [NVP98]. More recent work formalized ownership as a type system and showed how to increase its expressiveness [CNP01,BLS03]. Uniqueness was proposed as an aliasing construct by Minsky and later refined by Boyland and others [Min96,Boy01].

ArchJava's support for ownership and uniqueness is most closely based on the author's previous work on AliasJava. To date, AliasJava is the only ownership type system that has a publicly available implementation and substantial experience showing that the system is practical [AKC02]. AliasJava's ownership model was extended in a later paper to support multiple ownership domains per object and the detailed policy specifications described in section 2 above, providing both more expressiveness and stronger aliasing guarantees compared to previous ownership systems [AC04]. Policy specifications and multiple ownership domains are essential for modeling sharing constraints in software architecture.

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 uniqueness 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.

Acknowledgements

I would like to thank my thesis advisors Craig Chambers and David Notkin, as well as members of the language

and software engineering groups at the University of Washingtion and Carnegie Mellon University for their comments and suggestions. Thanks also to Brian Giovannoni for access to the Golden Gate application. This work was supported in part by NSF grants CCR-9970986, CCR-0073379, and CCR-0204047, the High Dependability Computing Program from NASA Ames cooperative agreement NCC-2-1298, the Army Research Office grant number DAAD19-02-1-0389 entitled "Perpetually Available and Secure Information Systems," and gifts from Sun Microsystems and IBM.

References

[AAG93] Gregory Abowd, Robert Allen, and David Garlan. Using Style to Understand Descriptions of Software Architecture. Proc. Foundations of Software Engineering, New Orleans, Louisiana, December 1994.

[AC04] Jonathan Aldrich and Craig Chambers. Ownership Domains: Separating Aliasing Policy from Mechanism. Proc. European Conference on Object-Oriented Programming, Oslo, Norway, June 2004.

[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 with ArchJava. Proc. European Conference on Object-Oriented Programming, Málaga, Spain, June 2002.

[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.

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

[Ald03] Jonathan Aldrich. Using Types to Enforce Architectural Structure. Ph.D. Thesis, University of Washington, August 2003. Available at http://www.archjava.org/.

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

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

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

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

[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.

[DRR+99] D. Dvorak, R. Rasmussen, G. Reeves, A. Sacks. Software Architecture Themes in JPL's Mission Data System. *AIAA Space Technology Conference and Exposition*, Albuquerque, NM,1999.

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

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

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

[ITU99] ITU-T. Recommendation Z.100, Specification and Description Language (SDL). Geneva, Switzerland, November 1999.

[LPZ02] K. Rustan M. Leino, Arnd Poetzsch-Heffter, and Yunhong Zhou. Using Data 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.

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

[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, California, October 1996.

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

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

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

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

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

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

[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.

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

[SN92] Kevin Sullivan and David Notkin. Reconciling Environment Integration and Component Independence. Trans. Software Engineering and Methodology 1(3):229-268, July 1992.