Using Types to Enforce Architectural Structure

Jonathan Aldrich
Carnegie Mellon University
jonathan.aldrich@cs.cmu.edu

Abstract

The right software architecture is critical to achieving essential quality attributes, but these qualities are only realized if the program as implemented conforms to its intended architecture. Previous techniques for enforcing architecture are either unsound or place significant limitations on either architectural design or on implementation techniques.

This paper presents the first system to statically enforce complete structural conformance between a rich, dynamic architectural description and object-oriented implementation code. We extend previous work to (1) explain what full structural conformance means in an object-oriented setting, and (2) enforce architectural structure in the presence of shared data. We show that the resulting system can express and enforce important structural constraints of an architecture, while still supporting key object-oriented implementation techniques. As a result of our conformance property, developers can be assured that their intended architecture is realized in code, so the system will exhibit the desired quality attributes.

1. Introduction

Designing the right software architecture [GS93, PW92] for a system is essential for achieving critical system-level quality attributes like evolvability, security, and performance [BCK03]. Just as important, however, is ensuring that the implementation conforms to the designed architecture. As a case in point, we worked with an enterprise that had just "bet the company" on a new product-line architecture, which was meant to enhance the agility, quality, and features of the company's products through enhanced reuse. The company's worst nightmare was that, after a huge investment in building the product-line architecture, engineers would be tempted to change to the code in ways that might be convenient for a single product but would degrade the overall architecture over time.

As this example shows, it is all too easy for a well-designed architecture to be subverted by a thousand little conveniences, each apparently harmless, but which together destroy the intended benefits of the architecture. Because it is crucial to achieving desired architectural properties in practice, conformance is a critical issue for *any* architecture. Better tools and methodologies are badly needed to enforce architectural design.

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 when run-time communication in the implementation conforms to architectural structure [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.

Effective static analysis approaches exist for checking communication integrity against a module view of architecture [MNS01, LR03]. Checking conformance to a run-time, component and connector view of architecture is much more challenging due to programming language mechanisms which support implicit communication, such as objects and mutable references. One approach eliminates these implicit communication mechanisms entirely [ITU99, Cha01], which works well for very static systems such as embedded control circuits. However, this approach is inapplicable to more dynamic systems and is unlikely to be accepted by practitioners used to the flexibility of object-oriented languages and design patterns. Other approaches postpone checks to run-time [Mad96], but this may be too late to avoid user-visible faults resulting from conformance problems.

We have been investigating an approach, ArchJava, which embeds an architectural description in code using new programming language constructs [ACN02]. As with any language-based approach, there are challenges to the industrial adoption of ArchJava. However, ArchJava is similar to model-driven engineering approaches which are of current interest in industry, and lessons learned may be applicable in these systems. More importantly, a direct expression of architecture in the programming language makes it feasible to, for the first time, enforce communication integrity with a dynamic, component and connector architectural view. Having solved the communication integrity problem in the simpler setting of ArchJava, in current work we are extending the approach to mainstream languages like Java [AA07].

Our previous work on ArchJava enforced communication integrity for function calls between components, but it did not enforce integrity for communication through shared data. This paper makes three primary technical contributions:

- We give the first precise technical definition for communication integrity in object-oriented systems, giving architects a way of thinking about conformance in object-oriented settings.
- We extend ArchJava to reason about communication through shared data, yielding the first approach that can statically enforce a run-time component and connector view of architecture in the general object-oriented implementation setting.
- We demonstrate that the resulting system can express architectural constraints that are directly relevant to quality attributes like performance and extensibility. Furthermore, the system supports challenging object-oriented coding idioms like the Factory and Observer design patterns.

Our approach uses ownership constructs from the AliasJava [AKC02,AC04] system to declare what data is conceptually a part of each component, as well as to describe data that is passed linearly from one component to another, or shared temporarily or persistently between components. Ownership allows us to express a more general architectural shared data connector abstraction compared to previous work [MQR95].

In the next section, we review the alias control constructs of AliasJava. Section 3 shows how these constructs can be integrated into ArchJava to support a specification of data sharing in an architecture. We show a number of architectures that illustrate common object-oriented implementation patterns, as well as highly dynamic architectural designs. Section 4 defines communication integrity precisely for ArchJava, explains how it is checked, and sketches a proof of conformance. Finally, section 5 discusses related work, and section 6 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. Aliasing Specification Model

The goal of AliasJava is to enforce 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 high-level aliasing constraints in the system architecture, then refine these constraints to specify aliasing within subsystems, modules, and individual objects.

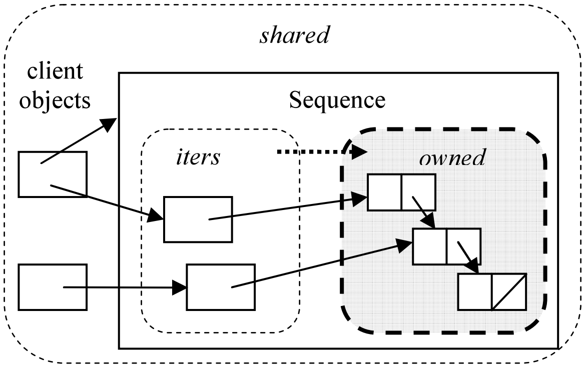


Figure 1. A conceptual view of the aliasing model used in AliasJava and ArchJava. The rounded, dashed rectangles represent ownership domains, with a gray fill for private domains. Solid rectangles represent objects. The top-level *shared* domain contains the highest-level objects in the program. Each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

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 *owned* 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 a dashed arrow in the diagram, allows objects in the first domain to access objects in the second domain.
- 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 reference 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.

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 `Sequence`'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 `SequenceIter` object (not shown), initializing the iterator to point to the first element of the linked list.

```
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, owned -> T.owner,
       iters -> T.owner;
  iters Iterator<T> getIter() {
    return new SequenceIter<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`.

Uniqueness and Lending. While ownership is 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 an 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. In previous work we 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.

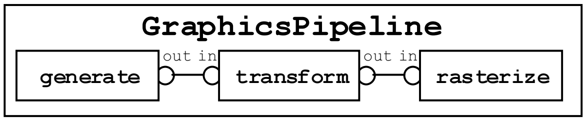
3. ArchJava

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 ownership 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 graphics pipeline. The generate component generates shapes to be displayed in the current scene. These shapes are passed on to the transform component, which applies the current transformation to each shape in turn. It then passes the shapes to the rasterize component to be drawn.

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. These invariants support important quality attributes, such as the ability to add and remove components from the pipeline, and the ability to use a concurrent thread in each component. As we introduce ArchJava through this example, we will discuss how these invariants are specified and enforced.



```
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: Generate 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.

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. 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.

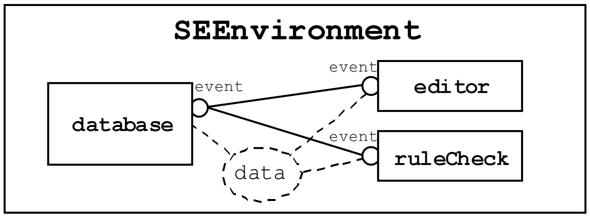
Each port declares a set of required and provided methods. A *provided* method is implemented by the component and is available to be called by other components connected to this port. Conversely, each *required* method is provided by some other component connected to this port. Each provided method 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 subcomponents 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.



```
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.

3.5. Example: Repository Architecture

Figure 4 shows the architecture of a software engineering environment. The architecture is structured as a repository, 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, using dependency injection to load third-party tools at run time. Therefore, the architecture declares a connect pattern between the event port of the database and the event port of the abstract component class Tool.

SEEnvironment reads a configuration file to determine the set of installed components and then instantiates them one by one using the instantiate-Tool 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 (determined at run time) to be created and linked into the software engineering environment. Thus ArchJava supports a level of dynamism similar to Darwin [MK96] but somewhat less dynamic than ArchWare [MKB+04], as ArchJava does not provide for changes to existing connections or the removal of components.

3.6. Shared Data Connectors

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. These shared ownership domains generalize the concept of shared variable connectors introduced in SADL [MQR95] to allow much richer forms of object-oriented interaction between components.

For example, 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 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, an important object-oriented idiom that is challenging to reason about in conventional implementation languages. ArchJava ensures that tool components in the SE environment can only communicate through event callbacks and through modifications to shared data, ensuring the efficient communication and ease of adding/modifying tools that are the quality attribute goals of the repository/mediator architectural style [SN92,GS93].

3.7. Implementation

An open-source compiler (based on Barat [BS98]) from ArchJava to Java bytecode is available for download at the ArchJava web site [Arc02]. ArchJava code can link with Java libraries, but communication integrity is only guaranteed if all code is run through the ArchJava compiler. Both typechecking and compilation are local, so when a source file is updated, only files dependent on its interface need be typechecked and recompiled. We have also implemented a tool that can generate ArchJava code from an architectural description in the Acme language, and a tool that automatically compares and synchronizes Acme and ArchJava architectures [AAN+06].

ArchJava's type system is as static as Java's: most checks are done at compile time, but run-time checks are performed at downcasts and array writes (the same places Java already does dynamic checks) to ensure that the domain parameters of an object match the parameters declared in the type of the cast or array.[1] Other papers provide additional details about the type system and the implementation techniques used in the compiler [AKC02,Ald03].

3.8. Experience

This paper focuses on how the extended ArchJava system is able to use types to capture architectural data sharing constraints, and thereby enforce full communication integrity. We have previously reported experience showing that ArchJava and AliasJava can be applied to nontrivial programs (10+ kLOC) with only moderate effort, and can provide benefits such as enforcing important architectural constraints, encouraging loose coupling, easing defect repair, and making communication more explicit [ACN02, AKC02, AAC07]. Our experience has also shown that capturing certain highly dynamic designs in ArchJava can be awkward, motivating our current work enforcing architecture in pure Java [AA07].

We have found ArchJava useful as a teaching tool, because it makes the often-abstract aspects of software architecture very concrete for students. Curricular material on using ArchJava to teach software architecture is available at the ArchJava web site [Arc02].

3.9. 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 critical to ensuring that a system achieves the benefits designed into the architecture. While the intuition behind communication integrity is straightforward, making this intuition precise in the presence of complex object-oriented designs is difficult. Here we define communication integrity in the ArchJava component model, but we believe the definition can be applied (with minor modifications) in broader settings as well.

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 in architectural domain `A` *accesses* (invokes a method or reads or writes a field of) a non-component object `B` which is in a different architectural domain.

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/Component `A` invokes a method on a `unique` component `B`, or
2. **Parent-child communication:** Object/Component `A` invokes a method on a component `B` 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`.

[1] Although ArchJava's type system is statically checkable, the language can still express dynamic architectures.

Discussion. Although the principle of communication integrity has a universally clear meaning, the way in which communication integrity is documented will vary from system to system. We believe that the definition above is appropriate for ArchJava because it permits only local communication between components—the essence of communication integrity—yet allows that local communication to occur through a number of important object-oriented patterns and idioms.

Each of the forms of communication above is essential in an object-oriented setting. Unique communication is important in order to allow implementations of loosely-coupled systems such as the pipeline example above. Although in principle we could have required parent-child communication to be done through explicit ports, our experience with ArchJava has shown that this would be awkward. Connection communication through explicitly declared ports is of course the standard architectural case for communication. Lent communication is necessary to support library code and efficient parameter passing, providing an escape hatch from the constraints of ownership while still allowing local reasoning about communication. Finally, shared domain communication supports communication through persistent shared objects, a common idiom in many object-oriented systems.

Our 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, we believe that the definition is a good compromise given the goal of supporting existing Java programs with few changes. Furthermore, we 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.

The author's dissertation 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 some other, non-`owned`, ownership domain. 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 part of the globally accessible `shared` domain. In practice, however, excessive communication through the `shared` domain makes reasoning more difficult, and so developers should avoid it, 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. A compromise would be to issue a warning when the `shared` domain is used.

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. Related Work

ArchJava. The initial ArchJava system enforced architectural conformance only for control flow between components, not for communication though shared data [ACN02]. Although our initial experience suggested that control-flow integrity is useful in practice, rigorous reasoning about general architectural properties requires understanding communication though shared data. This paper's extension of ArchJava to enforce communication integrity in the case of shared data is much more challenging, due to the ubiquitous and complex uses of shared data structures in object-oriented systems.

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 ownership 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].

Architecture Description Languages. A number of architecture description languages (ADLs) have been defined to describe, model, check, and implement software architectures [MT00]. The C2 system provides a framework for implementing software architectures, but does not automatically ensure that the code instantiating the framework respects architectural constraints [MOR+96]. 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.

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. The SPARK system takes a similar approach, supporting a subset of Ada without references in order to rigorously guarantee information flow properties [Cha01]. The prohibition of references is reasonable and even desirable for the telecommunications and other embedded systems for which SDL and SPARK were designed, but is inappropriate for the highly dynamic, object-oriented applications that ArchJava targets. 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. Our approach could be used to enforce conformance in these systems.

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]. Our uniqueness concept is based on Boyland's work [Boy01].

ArchJava's support for ownership and uniqueness is most closely based on the author's previous work on AliasJava, which includes 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].

No previous work, however, has applied ownership and uniqueness to the problem of architectural conformance. A contribution of this paper is showing how the concept of shared variable connectors, formally introduced in the SADL system [MQR95], can be generalized to shared ownership domains that allow rich object-oriented sharing relationships while retaining a strong guarantee of communication integrity. Our system also demonstrates that flexible policy specifications and multiple ownership domains are essential for modeling sharing constraints in software architectures.

References

[AA07] Marwan Abi-Antoun and Jonathan Aldrich. Compile-Time Views of Execution Structure Based on Ownership. International Workshop on Aliasing, Confinement and Ownership at ECOOP, July 2007.

[AAC07] Marwan Abi-Antoun, Jonathan Aldrich, and Wesley Coelho. A Case Study in Re-engineering to Enforce Architectural Control Flow and Data Sharing. Journal of Systems and Software 80(2), 240-264, 2007.

[AAN+06] Marwan Abi-Antoun, Jonathan Aldrich, Nagi Nahas, Bradley Schmerl, and David Garlan. Differencing and Merging of Architectural Views. Proc. of Automated Software Engineering, September 2006.

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

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

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

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

[BCK03] Len Bass, Paul Clements, Rick Kazman. Software Architecture in Practice. Addison-Wesley, 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.

[Cha01] Rod Chapman. SPARK – a state-of-the-practice approach to the Common Criteria implementation requirements. Proc. International Common Criteria Conference, 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.

[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.

[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.

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

[MKB+04] R Morrison, G. Kirby, D. Balasubramaniam, K. Mickan, F. Oquendo, S. Cîmpan, B. Warboys, B. Snowdon, and R. Greenwood. Support for Evolving Software Architectures in the ArchWare ADL. Proc. Working IEEE/IFIP Conference on Software Architecture, 2004.

[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.

[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 RealTime. 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.