ArchJava:
Connecting Software Architecture to Implementation

Jonathan Aldrich

Department of Computer Science and Engineering
University of Washington

Seattle, WA 98195 2350 USA
+1 206 616-1846

{jonal, chambers}@cs.washington.edu

ABSTRACT
Software architecture describes the structure of a system, and is useful for design, program understanding, and formal analysis. However, in existing systems an implementation may not conform to the designer's architecture, causing confusion, violating architectural properties, and inhibiting software evolution. ArchJava is an extension to Java that seamlessly unifies a software architecture with its implementation, ensuring that the implementation conforms to the architectural constraints. Therefore, programmers can visualize, analyze, reason about, and evolve architectures with confidence that architectural properties are preserved by the implementation.

Categories and Subject Descriptors
D.2.11 [**Software Engineering**]: Software Architectures – *Languages*, *ArchJava*; D.3.2 [**Programming Languages**]: Language Classifications – *Object-oriented languages*, *Java*; D.3.1 [**Programming Languages**]: Formal Definitions and Theory – *Syntax*, *Semantics*

General Terms
Documentation, Design, Languages, Theory, Verification.

Keywords
software evolution, program analysis, traceability, type system, communication integrity.

1. INTRODUCTION
Software architecture [GS93][PW92] is the organization of a software system as a collection of interacting components. A typical architecture includes a set of components, connections between the components, and constraints on how they interact. Describing architecture in a formal architecture description language (ADL) can make designs more precise and aid program understanding, implementation, evolution, and reuse. Many ADLs also support automated visualization and code-generation tools [SDK+95], specification and analyses of temporal properties including deadlock [AG97], formal reasoning about correct refinement [MQR95], and other tools and analyses.

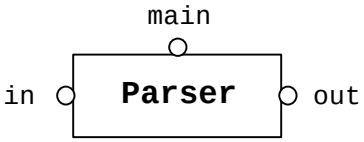
Existing ADLs, however, are not closely connected to an implementation language, causing key problems in the analysis, implementation, understanding, and evolution of software systems. Architectural analysis may reveal important properties of the architecture, but these may not be true in the implementation. One such property is *communication integrity* [MQR95], the constraint that components in the implementation may only communicate with the components they are connected to in the architecture. Some existing ADLs generate code that connects existing components [SDK+95], but place strong restrictions on the component code, unnecessarily burdening implementers. Other ADLs expect the implementation to be done manually in a different language, making it difficult to trace architectural features to the implementation for program understanding, and creating the danger that the architecture will become out of date as the implementation evolves.

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

- language and an implementation language, allowing flexible -
  architecture and implementation, and ensuring traceability
- ArchJava guarantees communication integrity in an architecture's implementation, even in the presence of advanced architectural features like dynamic component creation and connection.

The rest of this paper is organized as follows. The next section introduces the main features of ArchJava by a series of examples. Section 3 presents ArchJava's support for architectural design and evolution. Section 4 formalizes ArchJava's type system and outlines a proof of soundness and communication integrity in ArchJava. Section 5 describes techniques for compiling and visualizing ArchJava programs. Finally, section 6 discusses related work, and section 7 concludes.

2. THE ARCHJAVA LANGUAGE
This section introduces the ArchJava language by a series of examples. The language reference manual [AC01] gives the complete language semantics. ArchJava is a backwards-compatible extension to Java. It adds new language constructs to support *components*, *connections*, and *ports*.




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

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

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

Figure 1. A graphical parser component and its realization in ArchJava. The `Parser` component class uses three ports to communicate with other components in a compiler. The `Parser` receives `parse` messages from the body of the compiler through its `main` port. The `in` port declares a required method that requests a token from the lexical analyzer, and a provided method that initializes tokens in the symbol table. Finally, the `out` port requires a method that compiles an AST to object code, and provides a method that looks up tokens in the symbol table

2.1 Basic Components

A *component* is a special kind of object that uses only structured mechanisms to communicate with other components. Components are instances of *component classes*. Figure 1 shows an example of a parser component.

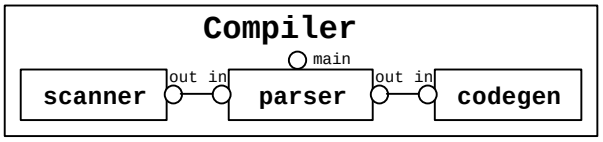
2.1.1 Ports

A component instance communicates with external components through ports. A *port* represents a two-way communication protocol between two component instances, from the point of view of one of the components. If a component participates in more than one logical communication channel, it specifies a port for each different channel. For example, the parser in Figure 1 declares three different ports: `main` represents communication with a top-level compiler component, `in` represents communication with a scanner component, and `out` represents communication with a code generator component.

Ports define two public interfaces [LHL77]. The *provided* interface is a set of public methods that can be invoked by other components connected to the port. The *required* interface is a disjoint set of methods that the component can invoke through the port. Required methods are implemented by the other components that a port is connected to. Method declarations within a port are labeled with the **`requires`** keyword or the **`provides`** keyword (optional) to distinguish which interface they belong to. For example, the `in` port of the parser defines a required interface containing the method `nextToken` and a provided interface containing the method `setInfo`.

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

In Figure 1, the `parse` method provided in the `main` port is implemented in a separate method definition; as syntactic sugar, it could also have been given a body inside the port. The `parse` method invokes the required methods of the `in` and `out` ports using regular method invocations, using the port as the receiving object. These calls will be forwarded to external components that implement the appropriate functionality. Required methods declared in different ports are potentially distinct; calling one of them results in invoking the provided method at the other end of that connection.

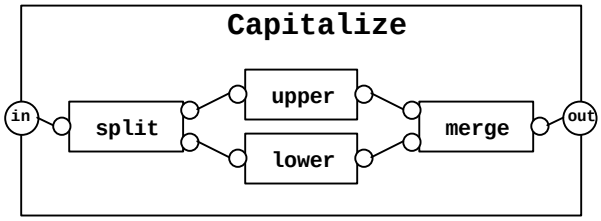



```
component class Compiler {
  void invoke(String args[]) {
    // for each file in args
    parser.main.parse(file);
  }

  Scanner scanner;
  Parser parser;
  CodeGen codegen;

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

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

2.2 Component Composition

In ArchJava, software architecture is expressed with *composite components*, which are made up of a number of subcomponents connected together. The subcomponents may themselves be composite components, enabling hierarchical architectures to be expressed.




```
component class Capitalize {
  Split s;
  Upper u;
  Lower l;
  Merge m;

  connect s.out1, u.in
  connect s.out2, l.in;
  connect u.out, m.in1;
  connect l.out, m.in2;

  connect in, s.in
  connect out, m.out;

  port in {
    requires char getChar();
  }

  port out {
    provides char getChar();
  }
}
```

Figure 3. A pipe-and-filter component that capitalizes every other letter in an input stream. The `in` port of the component is connected to the `in` port of a `Split` subcomponent that sends characters to alternating output streams. One stream goes though a component that capitalizes characters, the other goes through a component that converts characters to lowercase. A `Merge` component merges the two streams, and its output is connected to the output of the composite component.

Figure 2 shows how a compiler's architecture can be seamlessly expressed in ArchJava. This example shows the advantages of ArchJava—it is immediately clear that the parser communicates with the scanner using one protocol, and with the code generator using another. Furthermore, the architecture shows that the scanner does *not* communicate directly with the code generator. This kind of reasoning about the structure and communication patterns in a program can make a program understanding task easier.

2.2.1 Reasoning about Communication

If the drawing in figure 2 represented an abstract architecture to be implemented in Java code, there would be no way to verify the reasoning expressed above. For example, if the scanner was passed a reference to the code generator, it could invoke any method it wants to, violating the intuition communicated by the architecture. In contrast, programmers can have confidence that an ArchJava architecture represents an accurate picture of communication between components, because the compiler enforces communication integrity.

Communication integrity in ArchJava means that components in an architecture can only call each others' methods along declared connections between ports. Each component in the architecture can use its ports to communicate with the components to which it is connected. However, a component may not invoke the methods of its siblings in the architecture directly, because this would represent communication not declared explicitly in the architecture—a violation of communication integrity.

Communication integrity is enforced by ArchJava's type system: a component cannot get a typed reference to another component, and thus cannot invoke any of that component's methods (except the methods defined in `Object`). Classes may not declare fields or arrays of component type, and component types cannot appear in port interfaces, so references of component type cannot be passed between components. Finally, casts to component types are prohibited, so a component cannot downcast a reference of type `Object` to a component type.

2.2.2 Subcomponents

A *subcomponent* is a component instance that is declared inside another component class. Subcomponents have a lifetime equal to their *parent* component, and cannot directly communicate with components external to their containing component. Thus, communication patterns in ArchJava are hierarchical.

Subcomponents are declared using a *component field*—a field of component type inside a component class. Subcomponents are automatically instantiated when the containing component is created, and component fields are treated as **`private`**, **`final`**, and not **`static`**. Programmers can use a **`new`** expression in the field initializer in order to call a non-default constructor.

2.2.3 Connections

The **`connect`** primitive connects two or more subcomponent ports together, binding required methods to provided methods of the same name and identical signatures. Each port may only participate in one connection, so that it is clear where required methods are bound. For each required method, there must be exactly one corresponding provided method, but any number of required methods may be connected to a single provided method. Connections are instantiated just after the call to **`super()`** in the constructor of the enclosing component instance.

2.2.4 Calling Subcomponent Methods

According to the principle of hierarchical communication integrity, ArchJava allows components to invoke the provided methods of their subcomponents' ports. For example, the `invoke` method of the compiler example calls the `parse` method provided by the `main` port of the `parser` subcomponent.

2.2.5 Correspondence of Code and Architecture

Figure 3 shows a capitalization component with a pipe-and-filter architecture. Allen and Garlan use this somewhat contrived example [AG97] to point out that the architecture of a system may have a significantly different structure from its implementation, which in their paper is based on a top-down functional design. In ArchJava, the implementation of a system corresponds closely to its architecture, as can be seen by comparing the code for `Capitalize` to the abstract architecture drawing.

```
component class Compiler {
  public static void main(String args[]) {
    new Compiler().invoke(args);
  }

  // the rest of Compiler's implementation...
}
```

Figure 4. Creating a disconnected `Compiler` component

This feature has advantages and drawbacks. The big advantage is understandability—it is easy to see how the each feature in the architecture is implemented, and the architecture can be used to document the large-scale structure of the system. Furthermore, as requirements change, architecture and code will necessarily evolve together. The correspondence between architecture and code also allows the ArchJava compiler to easily verify key architectural properties such as communication integrity in the implementation. On the other hand, ArchJava limits the ways in which implementers can structure their system. Although object structures in ArchJava can be organized in flexible ways that cross architectural boundaries, components must obey the architecture's structuring and communication constraints in order to preserve communication integrity. Further experience will show if this tradeoff is worthwhile.

2.2.6 Connections to Subcomponents

As figure 3 demonstrates, a component may provide a method by connecting it statically to a provided method in a subcomponent, or a required method in some other port. Therefore, the connect statement can connect a containing component's port to one or more ports of its subcomponents, or can connect two ports of a single component together from inside the component. A component-local check is sufficient to verify that each provided method has exactly one implementation.

2.3 Dynamic Architectures

The constructs described above express architecture as a static hierarchy of interacting component instances, which is sufficient for a large class of systems. Static architectures are easy to reason about, visualize, and are amenable to finite tools such as model checkers. However, some systems require architectures that change dynamically, adding and removing components and connections. **Cite examples from papers in the literature.** The language features below extend the static constructs in a natural way, describing a static approximation of the ways a dynamic architecture can be instantiated at run time.

2.3.1 Disconnected Components

ArchJava allows a parent component to create *disconnected* child components dynamically with the **`new`** syntax. Like other components, disconnected components must follow the rules for communication integrity. Disconnected components cannot be connected to other components, so they can only call methods on their own subcomponents. Only the parent component can call a disconnected component's methods, as described above.

As a special case, disconnected components can be created at the top level with a null parent component. This mechanism can be used in the `main` method of a component-based application in order to create a top-level component and invoke its methods.

```
component class WebServer {
  Router r;
  connect r.main, main;

  dynamic Worker w;
  dynamic connect w.work, r.work;

  port main {
    provides Router.work newWorker() {
      Worker#w newW = new Worker#w();
      Router.work worker =
        connect(newW.work, r.work);
      return worker;
    }
  }
}

component class Router {
  dynamic port work {
    requires void job(Data data);
    provides void done() {
      freeList.add(sender);
    }
  }

  port main {
    requires work newWorker();
    provides void listen() {
      ...
      // when a HTTP request arrives
      work w;
      if (freeList.isEmpty()) {
        w = newWorker();
      } else {
        w =(work)freeList.remove(0);
      }
      Data data = ...;
      w.job(data);
      ...
    }
  }

  List freeList = new LinkedList();
}
```

Figure 5. Part of a web server component. The Router subcomponent accepts incoming HTTP requests, and manages a set of Worker components (not shown) that respond. A free list of workers available to handle requests is maintained; when the free list is empty and another request comes in, the Router requests a new worker on its main port. The WebServer then creates a new worker and connects it to the Router. The Router assigns jobs to Workers through the jobs port; when a worker finishes a job, the done() method adds the worker to the free list.

For example, Figure 4 shows the implementation of a compiler's `main` method.

2.3.2 Dynamic Components

A limitation of disconnected components is that they cannot be connected to other components. Furthermore, since array and object fields cannot have component type, and component casts are prohibited, dynamic components cannot be stored in aggregate data structures such as container classes. Dynamic components and connections overcome these limitations.

A *dynamic component field declaration* is an abstraction for zero or more components that may be created at runtime. It is declared with the **`dynamic`** modifier added to the regular component field declaration. The declaration does not actually create the subcomponents; instead, it allows the user to create one or more subcomponents at run time. Figure 5 shows a web server component that declares a dynamic field `w` of component type **`Worker`**.

Dynamic components can be instantiated at run time in the static scope of the component that declared the dynamic component field. Each dynamic component instance is associated with a particular dynamic component field declaration. This association is captured in a *component field type*, which combines a component type with a field name using the # operator. The component field type is specified in the **`new`** expression when instantiating dynamic components. Variables can also be given a component field type. The `newWorker` method in Figure 5 creates a component of type `Worker#w` and assigns it to a local variable of the same type.

Dynamic components can be stored temporarily in an aggregate data structure, retrieved, and then downcast to a component field type within the static scope of their parent component. The downcast includes a run time check to verify that the component's parent is equal to **`this`**, and that the component's *parent field* is equal to the field in the component field type. This check ensures that only a dynamic component's parent can invoke its methods directly, and that dynamic component field declarations are properly matched with component instances.

2.3.3 Dynamic Connections

Dynamic components can be linked together using dynamic connections. A *dynamic connection declaration* is an abstraction for zero or more connections may be created at runtime between the component ports specified. The dynamic connection can be instantiated at runtime with a *connect expression*. Connect expressions are passed the ports to be connected as arguments and return a *connection object* that represents the set of connected ports.

In order to guarantee communication integrity, for each connect expression in the program, there must be a dynamic connection declaration connecting matching ports. For instance, in Figure 5, a dynamic connection is declared with **`dynamic connect`** `w.work, r.work` and is instantiated with the expression **`connect`**`(newW.work, r.work)`. Here `newW.work` matches `w.work`, since `newW` has a component field type `Worker#w` that includes field `w`, so the connection expression can be statically checked for conformance to the dynamic connect declaration.

2.3.4 Dynamic Ports

Often a single component communicates with several other components using the same conceptual protocol. For example, the `Router` component in the web server communicates with a dynamically varying set of `Worker` components. Thus, there must be a way to connect the `work` port to multiple components. When the router invokes a worker's methods through its work port, there must also be a way to specify which worker is intended to respond to the message send.

A *dynamic port* is a port that can participate in more than one connection simultaneously. This allows a component instance to communicate with multiple other dynamically created component instances at run time. A dynamic port can be connected to other ports using dynamic connections or static connections.

2.3.5 Required Interfaces

Required methods can be invoked through dynamic ports by calling through a port's *required interface*. Each port defines an interface that includes all the required methods in that port. When a dynamic connection is instantiated, the connection object returned from the connect expression implements the required interface of all the connected ports; thus its type is a union of all the ports' required interfaces. This union type cannot be written directly in Java, but can be assigned to a variable of the appropriate required interface type. In Figure 5, the connection object is a subtype of the interfaces `Router.work` and `Worker.work`, so it can be assigned to a variable of interface type `Router.work`.

Provided methods in a dynamic port can find out which connection a particular call came from using the `sender` keyword. Defined in every provided method, the `sender` variable is a connection object that implements the required interface type of the port the call was made from. Some methods may be provided in multiple ports; in this case, `sender` has static type `Object` and the programmer must use casts or **`instanceof`** tests to get the appropriate interface type. Since some methods may be called either through a port or though a direct call, the `sender` variable is null when a direct call was made. In Figure 5, the `sender` variable is used by the `done` method of the `Router` to cache a connection for use in a later HTTP request.

When the connection object is read from a collection (as in `freeList.remove`) it can be cast to the appropriate required interface type so that methods can be invoked through the connection. To enforce communication integrity, ArchJava must ensure that connection objects cannot be used by components that were not involved in the original connection, because this might violate the communication structure specified by the dynamic connection declarations. This is verified by a run-time check in each call that ensures that the calling component is one of the connected component instances that requires the called method (and throws an `IllegalConnectionException` if not).

2.3.6 Removing Components and Connections

Just as Java does not provide a way to explicitly delete objects, ArchJava does not provide a way to explicitly remove components and connections. Instead, components are garbage-collected when they are no longer reachable through direct references or connections.

2.4 Benefits of ArchJava

The ArchJava web server implementation has a number of significant advantages over a similar Java implementation. The architectural connections between objects are explicit, while in Java they must be derived from the code using abstract interpretation and alias analysis. The explicit, hierarchical architecture makes visualization of program structure easy, whereas visualization of "object soup" structures in Java can be

```
component class Parser {
  port main {
    provides abstract void
      parse(String file);
  }
  port in;
  port out;
}
```

Figure 6. Early Parser Design

very challenging. Communication integrity is guaranteed, allowing better reasoning about communication patterns. Subcomponents are not just private, but encapsulated—references cannot usefully escape the enclosing component. The static declarations of dynamic connections also provide a novel way to reason about dynamic object structures, a task that is difficult in Java.

ArchJava also provides other, less central, advantages. Dependencies between components are explicit, supporting looser binding between components and promoting better understanding of what services a component requires. The looser binding combined with the explicit architecture makes it easy to swap in new components, or interpose a "smart connector" that performs buffering or interface translation between two components. This task can be more difficult in Java, since it involves changing the object creation expression, necessitating design patterns like factory methods and decorators [GHJ+94]. The port mechanism allows programmers to explicitly specify the different communication protocols that a component participates in. Finally, ArchJava was designed to improve automated reasoning by tools such as model checkers and alias analyses. For example, static architectures are a natural fit for model checkers, and dynamic connection declarations could help an alias analysis to track references more effectively. Future experience building tools will determine if the design meets this goal.

In summary, ArchJava provides a number of features that help developers to express, visualize, and reason about software architecture.

3. DESIGN AND EVOLUTION

In addition to supporting fully developed architectures, ArchJava supports incomplete architectures during the design phase, and supports architectural evolution through component subtyping and structural reconfiguration as requirements change.

3.1 Architecture Design

One of the advantages of formal architecture description languages is that designers can get feedback on their design before it is implemented. Existing ADLs support architectural typechecking, deadlock detection, conformance to architectural styles, and other analyses. As a result, architects can easily experiment with different architectures at design time, catching design errors earlier than would otherwise be possible.

ArchJava provides support for constructing, visualizing, and typechecking incomplete architectures. This allows architects to visualize their architecture and ensure key static and dynamic properties during design, helping them to find architectural errors earlier.

```
component class ByteCodeCompiler
          extends Compiler {
  CodeGenerator cg = new ByteCodeGenerator();
}

component class ByteCodeGenerator
          implements CodeGenerator { ... }
```

Figure 7a. Compiler with a byte-code generator back end

```
component class LoggingWorker
          extends Worker { ... }

component class LoggingWebServer
          extends WebServer {

  Router.work newWorker() {
      Worker#w newW = new LoggingWorker#w();
      ...
   }
}
```

Figure 7b. A logging web server that uses a `LoggingWorker` component subclass

The key design advantage of ArchJava over existing ADLs and implementation languages is that it supports iterative development and checking. As the architecture and implementation is gradually fleshed out, the ArchJava compiler will incrementally check the consistency of the architecture and implementation. Unlike previous systems, designers can be confident that their code will conform to their architectural design at every stage of the implementation.

The ArchJava compiler can be run in complete mode (the default) or incomplete mode. The complete mode verifies that all components are implemented, while the incomplete mode is tolerant of partial implementations. A component can be left incomplete simply by omitting subcomponents and connections. A port's body can be also left out, signifying that the port's interface is not yet specified. No interface checking is done when an incompletely specified port is connected to another port. As in Java, the body (not the signature) of a method can be left unspecified by using the **`abstract`** modifier.

3.1.1 Example IV: Parser Design

Figure 6 shows an early design of the `Parser` component. The `Parser` can be connected to the other components in the compiler architecture as described in Figure 2. The designer has left out the bodies of the `in` and `out` ports, so no interface checking is done when these ports are connected to other ports. Furthermore, no definition is given for the **`abstract`** `parse` method, but its signature can be checked against the call in `main()`.

3.2 Evolution through Inheritance

Like other object-oriented languages, ArchJava supports software evolution through inheritance and subtyping. As with regular classes, component classes can extend another component class. Components cannot extend ordinary classes (other than `Object`) or implement ordinary interfaces; this would allow violations of communication integrity, since components could be used as the interface or class type by components that cannot be communicating directly.

```
component class Split {
  port in {
    requires char getChar();
  }
  port out1 {
    provides char getChar();
  }
  // similar for out2
}

component class NewSplit {
  port in {
    requires char getChar();
  }
  port out1 {
    requires void putChar(char c);
  }
  // similar for out2
}
```

Figure 8. The interfaces of old & new `Split`

Inheritance works in the expected way; existing fields, methods, ports, component fields, and connections are inherited from the component superclass. As in Java, methods can be overridden, but component subclasses can also override ports and component fields.

An overriding port must specify a superset of the original port's provided method signatures, in the same way as Java allows subclasses to define additional methods. However, overriding ports must have an identical set of the original port's required methods, because inherited code may depend on the entire required interface, and existing connections cannot provide any additional required methods. Component subclasses may also declare additional ports as long as they do not contain any required methods.

Component fields can be overridden with a component field of the same name and type, but a new initialization expression. New components and connections can also be added in a subcomponent.

3.2.1 Subtyping

Instead of inheriting the interface *and* the implementation of another component class, a component class may inherit only the interface. This represents subtyping without inheritance, allowing one component class to be used in place of another even if there is no implementation inherited. Unlike Java, there is no way to implement multiple interfaces—either a component class extends a single other component class, or it implements the interface of a single other component class. These rules are necessary to ensure component substitutability, which is trickier than in Java because of required methods.

When a component implements the interface of another component, it must implement all of the public and package methods of that component (including methods provided by ports). Ports can be overridden as discussed above. Component fields and connections are not inherited in interface inheritance, but the implementing component class can specify new component fields and connections.

3.2.2 Examples of Evolution using Subtyping

The checked subtype relation can be used to increase confidence that a program evolution step is correct. In Figure 7a, the

```
component class NewCapitalize {
  NewSplit s;
  Upper u;
  Lower l;
  Buffer b1;
  Buffer b2;

  connect in, s.in;
  connect s.out1, b1.in;
  connect b1.out, u.in;
  connect s.out2, b2.in;
  connect b2.out, l.in;

  port in {
    requires char getChar();
  }
  // rest of component...
}

component class Buffer {
 port in {
  synchronized void putChar(char c);
 }
 port out {
  synchronized char getChar();
 }
 // buffering code...
}
```

Figure 9. Capitalization Component with `NewSplit`

`Compiler` component from Figure 2 has been subclassed to include a code generator that generates bytecode instead of executable code. The new component class is specified in an explicit initialization expression. It is a subtype of the `CodeGenerator`, ensuring that its interface matches the code generator's interface.

Subtypes can also be used for dynamic components and connections. Figure 7b shows a component that extends the web server from Figure 5 to instantiate a `Worker` subcomponent that logs the jobs that it performs. Because `LoggingWorker` inherits from `Worker`, it could be mixed freely with `Worker` objects in the same program. Subtyping relationships between component types extend to component field types with the same field name; thus `LoggingWorker#w` is a subtype of `Worker#w`, allowing assignment and casts to type `Worker#w`.

3.3 Evolution through Reconfiguration

Because ArchJava makes connections between components explicit, it can support evolution through component reconfigurations. Such reconfiguration can be more difficult in Java because a conceptual connection may be spread out in many references throughout the program, all of which must be potentially changed by an evolutionary step.

Consider the capitalization example from Figure 3. Assume that a better `Split` component has been identified, and the designer wants to replace `Split` with `NewSplit`, shown in Figure 8. This presents a tricky evolutionary step, because (as is often the case in real component development) there is no subtyping relationship between `Split` and `NewSplit`. The out port contains a different method name and signature. Furthermore, the components have a different control model—the original component is passive, but the new component contains an active thread that requests characters at the `in` port and sends them on the `out` port.

This evolutionary step might be difficult in Java, because a programmer would have to track down all the uses of the old component and change them to work appropriately with the new component. Alternatively, the programmer could trace all uses of the old component to the creation point, and write a wrapper for the new component. Both of these approaches are potentially challenging and error-prone (especially in more complex examples), simply because the communication patterns between components are often spread out and component aliases are difficult to track in the source code.

Figure 9 shows one possible solution to this interface-matching problem in ArchJava. A synchronized buffer component provides the `putChar` method that is required by the `NewSplit` component, and provides the `getChar` method that is required by the `Upper` and `Lower` components that the old `Split` component was connected to. Two buffer components are used to connect the `NewSplit` component to the existing `Upper` and `Lower` components. This is an example of how the mediator design pattern [GHJ+94] can be applied in ArchJava to allow mismatched interfaces to communicate.

4. ARCHJAVA FORMALIZATION

4.1 Definition of Communication Integrity

Communication integrity is the key property of ArchJava that ensures that the implementation does not communicate in ways that could violate reasoning about architectural properties. Intuitively, communication integrity in ArchJava means that component instance `A` cannot call the methods of component instance `B` unless `B` is `A`'s subcomponent, or `A` and `B` are sibling subcomponents of a common component instance that declares a connection between them.

To define communication integrity more precisely, we first define the dynamic execution scope of a component to include all a component's methods and the methods they transitively invoke until a method of another component is called:

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

Now we can define communication integrity:

Definition 2 [Communication Integrity for ArchJava]: A program has communication integrity if, for all method calls `b.m(...)` in an executing method frame `mf`, where `b` is a component instance and `mf` ∈ *escope*(`a`), *either:*

1. `a` = `b`, *or*
2. *parent*(`b`) = `a`, *or*
3. *parent*(`b`) = *parent*(`a`) ∧ "`dynamic connect f1, f2,...`" ∈ *class*(*parent*(`b`)), where `f1` = *parentfield*(`a`) ∧ `f2` = *parentfield*(`b`)

In this definition, we assume `m` is not one of the methods defined in `Object`; these methods are not very interesting for reasoning about component communication, and our system does not enforce communication integrity on them.

Also, notice that there can be communication aside from method calls due to data sharing from aliased objects, static fields, and the operating system. Existing ways to control this form of communication often involve significant restrictions on programming style. ArchJava's design allows reasoning about communication through method calls between components, while imposing few restrictions on programmers. Future work includes developing ways to control these additional communication channels while preserving expressiveness.

4.2 Enforcement of Communication Integrity

There are three ways a component method could be invoked in a way that violates communication integrity. First, a method could be invoked on an expression of component type. Second, a method could be invoked on an expression of component field type. Finally, a method could be invoked on a connection object through a port's required interface.

These three possible violations of communication integrity are prohibited by a number of static and dynamic checks ensuring four key invariants:

Invariant 1 [Component Type Encapsulation]: Inside *escope*(`A`), all expressions of component type refer either to `A` or to a disconnected component created in *escope*(`A`).

Invariant 1 is enforced by a number of restrictions imposed by the ArchJava language. First, no fields in objects may be given a component type—only *component fields* in components. Second, casts to component type are illegal. Third, arguments to public component methods may not have component type, and non-public component methods may only be called on **`this`**. Because of these restrictions, the only way to get an expression of component type in ArchJava is to create a disconnected component (whose conceptual parent is `A`) or to use **`this`** inside a component's lexical scope. These expressions can be passed around to private component methods and to object methods within a component's execution scope, but must be implicitly downcast to `Object` before escaping the component's execution scope. Invariant 1 enforces communication integrity in the first case, where a method is invoked on an expression of component type.

Invariant 2 [Component Field Type Encapsulation]: Inside *escope*(`A`), all expressions of component field type with field `f` refer to a subcomponent of `A` that was associated with the component field `f` at creation.

Invariant 2 is enforced in a similar way to invariant 1. Casts to a component field type are allowed in ArchJava, but the cast dynamically checks that the target is a subcomponent of **`this`**, and the target's parent field is the one specified in the type. The only other way to get an expression of component field type in ArchJava is to create a subcomponent (with parent `A` and parent field `f`) or to refer to a component field inside a component's scope. Invariant 2 enforces communication integrity in the second case, where a method is invoked on an expression of component field type.

Invariant 3 [Connection Creation Integrity]: All connection objects created dynamically in a component instance `A` connect subcomponents of `A`, such that the corresponding fields were connected in a dynamic connect declaration in `A`.

```
component class C {
  dynamic SubC s;
  void foo(Object o) {
    Object o = new SubC#s();
    ((SubC#s)o).bar();
  }
}
```

Translation:

```
interface C {
  void foo(Object o)
}

class C$def extends $Component$ implements C
{
  C$tag$s s = new C$tag$s();

  C$def(Object $parent$) {
    super($parent$);
  }

  void foo(Object o) {
    Object o = new SubC$def(s);
    SubC temp$1 = (SubC)o;
    if (s != temp.$parent$)
      throw new ComponentCastException();
    temp$1.bar();
  }
}

class C$tag$s { }
```

Invariant 3 is enforced by typechecking connect expressions. The typechecker ensures that each expression being connected is of some type `f.C`, where `f` is a field of `A` with a type `D` and `C` is a subtype of `D`, and the fields in the expression types are connected in a dynamic connect declaration. The type check, combined with invariant 2, ensures invariant 3.

Invariant 4 [Connection Invocation Integrity]: For all invocations of a required method through a connection object `C` (or a static port `P`) in the scope of a component instance `A`, `A` is one of the objects connected by `C` (or connected to `P`).

Invariant 4 is enforced with a dynamic check done by the dynamically created connection objects, and by the static semantics of static connections. Together, invariants 3 and 4 enforce communication integrity in the second case, when a method is invoked on a connection object through a port's required interface.

4.3 Formalization as ArchFJ

To prove communication integrity and the safety of our type system, we have formalized the core of ArchJava as ArchFJ. ArchFJ is based on Featherweight Java [IWP99]. This section defines the syntax, types, subtyping rules, dynamic semantics, and static semantics for ArchFJ. We outline the proofs of communication integrity, subject reduction, and progress.

To be inserted from other document

5. TOOL SUPPORT

Tool support for ArchJava includes planned compilation, analysis, and visualization tools.

5.1 Compilation Technique

In this section we outline a proposed compilation technique for ArchJava. Our system allows separate compilation, although a global compilation system could produce more optimized code. The details can be found in an accompanying technical report [AC01]. **Should we really be citing this? It's a bit out of date right now…**

5.1.1 Components

A component class is translated into an interface containing all of the methods that the component provides, and a class that implements the interface and defines the methods. If the component class inherited from `Object`, the generated class inherits from `$Component$`, an internal class that has a `$parent$` field to keep track of the component's parent. The static component fields are translated into private final fields of the same type, initialized just after the call to `super()` in the containing component's constructor. Each port is translated into a new interface, containing all of the port's required methods. In addition, static ports generate a private final *port field* of the interface type. Calls to required methods through that port are converted to invocations on the object in the port field. A *sender argument* of the port's required interface type is added to all provided methods, in order to implement the `sender` keyword. If the same method is provided in multiple different ports, the sender will be of static type `Object`, and must be cast to the appropriate type in the ArchJava source code.

Each component field generates a new *tag class* that encodes the field name, and a private final field initialized to an object of the tag class. The value in the tag field is later passed as an extra *parent tag* argument to the constructor of subcomponents, where it is passed to the `$Component$` constructor and assigned to a private final *parent tag field*. This value is checked when an object is downcast to a component field type—the downcast is translated into a call to a downcast method, which verifies that the value in the current component's tag field is equal to the parent tag of the object being downcast. **Figure X10** shows an example of translating a dynamic component.

5.1.2 Connections

Each static and dynamic connection declaration generates a new class that implements the required interfaces of the connected ports. The connection's constructor takes the connected components as arguments and assigns them to internal final fields. Dynamic connections are instantiated at dynamic connect expressions, and static connections are instantiated in the constructor of the component class containing the connection.

In order to verify that calls through a connection are made from one of the connected components, all required methods have an extra sender argument generated and provided by the compiler. When a required method is called, `this` is passed as the sender. For dynamic connections, the implementation methods in the connection object check that the sender is one of the connected objects that requires the given method. The connection object then forwards the method call to the connected component that provides it, passing the connection object itself as the `sender` so that the invoked method knows which connection it was invoked from. **Figure X11** shows an example of translating a dynamic connection declaration.

A provided method that is implemented by a connection to subcomponents or to another required method is given a body that simply calls the appropriate method.

In addition to generating code, the compiler must also do a number of semantic checks. The most important of these are the communication integrity checks, and the well-formedness checks for connections. The latter verifies that there is a unique provided method for each required method in the connected ports.

5.1.3 Performance

The overhead of ArchJava programs compared to ordinary Java implementations includes indirection through connection objects and the extra check when casting to a component field type. Assuming a JIT compiler, all of these incur a cost roughly comparable to a dynamic message dispatch.

We theorize that the architectural constructs of ArchJava will me most useful at the larger scales of the application, meaning that most method calls will use ordinary dispatch and not experience any ArchJava overhead at all. Therefore, we expect that performance is likely to be indistinguishable from a comparable Java implementation, or at worst will be comparable to the existing overhead of object constructs.

5.1.4 Concurrency

Concurrency is largely orthogonal to ArchJava. Components can protect themselves from concurrent access by using synchronized methods or other means, as in Java. Since the structures generated by the compiler are all immutable, data races are not an issue.

5.2 Visualization

Because of the structural nature of software architecture, is it particularly important to provide tools that can visually display, navigate through, and edit the architecture. As an initial, proof of concept implementation, we are defining a translation from ArchJava to Acme [GMW97], an architecture interchange language. This will allow ArchJava architectures to be browsed and edited in the AcmeStudio environment [Kom98], a freely available tool for Acme architectures. An additional advantage of translations to Acme is that ArchJava architectures can be compared with architecture descriptions written in other languages, and analysis tools written for other ADLs can potentially be applied.

```
component class Component {
  dynamic SubComponent1 s1;
  dynamic SubComponent2 s2;
  dynamic connect s1.p1, s2.p2;
}

Translation:

  // compiler-generated connection
class Component$Connect1 {
  implements SubComponent1$port$p1,
             SubComponent2$port$p2 {

  Component$Connect1(Object s1, Object s2) {
    this.s1 = s1;
    this.s2 = s2;
  }

    // required by port p1 in SubComponent1
  public final int foo(int arg1,
                       Object sender) {
    if (sender != s1)
      throw new
       IllegalConnectionException();
    return s2.foo(arg1, this);
  }

  private final Object s1;
  private final Object s2;

  ...
}
```

Figure X11. Translation technique for dynamic connections

6. RELATED WORK

6.1 Object-oriented Programming Systems

Software Architectures can be implemented in object-oriented languages such as C++ [**ref**] and Java [GJS97]. There are a number of differences between an architecture expressed in an object-oriented language and an ADL. First of all, object interfaces declare the services an object provides, but do not specify the services the object relies on. This information must be gleaned from the details of the implementation. Second, there is no direct way to specify a hierarchical communication structure, only local connections between objects. This makes structural visualization and analysis hard, impeding understandability. Finally, there is no concept of communication integrity, beyond the constraint that calls obey the type system. Any object can communicate with any other object, if it has a reference and casts it to an appropriate type. In contrast, ArchJava supports reasoning about communication between component *instances*, even in the presence of aliasing.

UML [**ref?**] is a notation for expressing object-oriented designs. It is capable of expressing relationships between objects that in some cases go beyond what ArchJava provides, for example, in specifying the multiplicity of object relationships. Although

tools to generate code from UML have been built [refs?], the notation has no inherent semantic base, a problem that is the subject of current research [ref?]. UML also has no concept of communication integrity or hierarchy, making it difficult to visualize and reason about communication patterns.

Component-based infrastructures such as COM [ref], CORBA [ref], and JavaBeans [ref] further support development of applications from components. Some tools for JavaBeans even support graphical ways to connect components together, allowing simple architectures to be visualized. Possible drawbacks in such systems include poor support for structural specification of dynamically changing systems, no concept of communication integrity, and a complex mapping from component events and properties to the code that implements them. On the other hand, these infrastructures often support multi-language, distributed systems.

Tools such as Reflexion Models [MNS95] have been developed to enable programmers to visualize the structure of existing programs. These tools are particularly effective for legacy systems, where rewriting the application in a language that supports architecture directly would be prohibitively expensive. Specifying architecture directly in ArchJava has the advantage that architecture is visible in the source code, and ArchJava allows reasoning about the connections and communication integrity between component instances, not just component classes.

6.2 Advanced Type Systems

Advanced, polymorphic type systems such as those found in ML [ref], GJ [ref], and Cecil [ref], allow more specific specification of types, but still cannot effectively express program structure.

More recent work proposes type systems for controlling aliasing in object-oriented programs that can be used to enforce a kind of communication integrity. Pointers that are aliased by two components can create a back door for communication that does not flow through a declared communication channel. Linear types can be used to declare unique objects that are unaliased [Min96]. Passing a unique object from one component to another does not create an aliasing problem, since the source component may not use the object again.

Other research has investigated enclosing types. Early work such as Islands [ref] or Balloons [ref] imposed strict rules on sharing objects between components. These systems can guarantee communication integrity, but also limit expressiveness. More recently, Flexible Alias Protection [NVP98] strikes a balance between guaranteeing aliasing properties and allowing flexible programming idioms. Although these systems aid in reasoning about aliasing, they do not directly express software architecture.

Techniques like these are the subject of ongoing research, and we plan to incorporate them into ArchJava in the future. Our current strategy of guaranteeing communication integrity of calls between components allows considerable reasoning about software architecture while putting minimal restrictions on implementers. With linear and/or enclosing types, ArchJava could still allow flexible implementation strategies, and guarantee a stronger version of communication integrity: that data does not flow between components except along declared communication channels.

6.3 Advanced Module Systems

Advanced module systems such as MzScheme's Units [ref] and ML's functors [ref] can be used to describe the static architecture of a system. The FoxNet project [ref] shows how functors can be used to build up a network stack out of statically connected components.

Using a powerful module mechanism for architectural connections has the advantage of using an existing language construct. However, using an explicit architectural description also has several advantages. First, programmers and automated tools can more easily recognize explicit architectural constructs, while it may be hard to distinguish between modules used to encapsulate an ADT and modules used to link up conceptual components. Thus, an explicit architecture enhances visualization and automated reasoning about architecture. Second, module systems do not define, document, or enforce communication integrity—this must be done using program annotations and the discipline of implementers. For example, if a module contains free variables or references to internal data structures of other modules, these can be used as communication "back doors." Finally, modules cannot be used to define or reason about object instances in dynamically changing architectures, because module connections are static. Instead, the ordinary type system must be used, with the drawbacks discussed above.

6.4 Existing Architecture Description Languages

A number of Architecture Description Languages have been defined to describe, model, check, and implement software architectures [MT00]. Many of these languages were partly inspired by the hardware architectures expressible in VHDL [ref?].

Many ADLs support sophisticated analyses. For example, Wright [AG97] allows architects to describe detailed communication protocols in a language based on CSP. Connections can be checked for compatibility, deadlock, and other properties. SADL [MQR95] formalizes architectures in terms of theories, shows how generic refinement operations can be proved correct, and describes a number of flexible refinement patterns.

Acme [GMW97] is an architectural interchange language. It defines a basic syntax for components and connections, and has extensible properties that can express the features of other languages. xArch [ref] is a new, similar project based on XML [ref]. Automated translators have been defined from Wright [AG97] and Aesop [ref] into Acme, so developers can compare architectures and take advantage of a set of common tools. We are defining a translation from ArchJava to Acme to leverage existing visualization and analysis tools.

Rapide [LV95] is an ADL designed to support event-driven simulations of software architectures. An executable sub-language includes reactive constructs for event-driven programming, as well as more conventional object-oriented and procedural constructs. Although Rapide was primarily designed to support architecture simulation, components in a Rapide architecture can be given real implementations in the executable

sub-language or in languages such as C++ or Ada. Rapide supports declarative event-based dynamic connections that support precise reasoning about event traces and dependencies. In contrast, ArchJava's dynamic connection declarations describe communication structure precisely, allow reasoning about component instances, and fit naturally into conventional object-oriented languages.

There are *style guidelines* that can be used to help assure communication integrity in Rapide [LV95]. In particular, components must only communicate with other components through their own interfaces, and interfaces cannot include references to mutable types. These guidelines are not enforced automatically, and are incompatible with many common programming idioms, such as shared data structures. ArchJava allows a more flexible programming style with objects, while still automatically enforcing communication integrity among components. The execution trace of an implementation in a language such as Ada can also be checked for structural and event conformance to Rapide architectures [Mad96].

UniCon [SDK+95] and other ADLs generate code to connect components together. The components themselves must be implemented in other languages. This allows easy incorporation of legacy code, but can restrict their interfaces to what the code generation system can support, and also opens the door to violations of communication integrity. The C2 system [MOR+96] provides runtime libraries in C++ and Java that connect components in as specified in the C2SADEL ADL. C2 requires a particular architectural style, which can enhance reasoning but also restricts the use of idioms like shared data structures and limits the scope of C2's applicability. Communication integrity relies on programmers following C2's implementation guidelines.

7. FUTURE WORK AND CONCLUSION

There are a number of possible directions for future work in integrating programming languages and software architecture:

- Supporting distributed software architectures with components implemented in multiple languages
- Applying temporal specifications to components during design and model-checking the implementation to verify that it conforms to the architecture
- Applying linear and enclosing types to enforce a dataflow version of communication integrity
- Developing tools like Reflexion Models to support migration from Java to ArchJava
- Applying ArchJava to large real applications, to test how well it can express architecture and support program evolution

In conclusion, ArchJava allows programmers to effectively express software architecture and then seamlessly fill in the implementation with Java code. At every stage of development and evolution, programmers can have confidence that the implementation conforms to the specified architecture, because ArchJava enforces communication integrity. Therefore, ArchJava enables effective design, better program understanding, and a cleaner evolutionary path than existing alternatives.

8. ACKNOWLEDGMENTS

Thank members of the Cecil group and others for their feedback.

Craig's funding

Notkin (if not co-author)

Appendix A: ArchJava Grammar Extensions

We add the following new keywords to Java:

```
component
port
provides
requires
dynamic
connect
sender
```

The following grammar elements have been added to the Java grammar described in the Java Language Specification [GJS97]:

```
type_decl ::= ...
          |   component_decl

component_decl ::=
    modifier* component class id^type component_body

component_body ::= { component_body_decl* }

component_body_decl ::= class_body_decl
                    |   port_decl
                    |   connect_decl


port_decl ::= modifier* port id port_body

port_body ::= { port_body_decl* }

port_body_decl ::= provides_opt method_declaration
               |   requires method_header ;

connect_decl ::= dynamic_opt connect port_list;

port_list ::= name
          |   port_list , name

method_invocation ::= ...
                  |   connect_expression

connect_expression ::= connect ( argument_list )

primary_no_new_array ::= ...
                     |   sender

qualified_name ::= ...
               |   name # id
```

9. REFERENCES

[AC01] Jonathan Aldrich and Craig Chambers. The ArchJava Language and Runtime System Reference Manual. Available at: http://www.cs.washington.edu/homes/jonal/archjava/

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

[GJS97] Gosling J., Joy B., & Steele G., *The Java Language Specification*, Addison Wesley, 1997.

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

Oriented Software. Reading, Massachusetts: Addison-Wesley, 1994.

[GMW97] David Garlan, Robert T. Monroe, and David Wile. Acme: An Architecture Description Interchange Language. Proceedings of CASCON '97, November 1997.

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

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

[Kom98] Andrew Kompanek. AcmeStudio User's Manual. Tools and manual available at http://www.cs.cmu.edu/~acme/.

[LHL77] Lampson, B.W., Horning, J.J., Mitchell, J.G. and Popek, G.J. *Report on the Programming Language Euclid*, ACM SIGPLAN Notices 12, 2 (February 1977), 1-79.

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

[Mad96] N. Madhav. Testing Ada 95 Programs for Conformance to Rapide Architectures. In Proceedings of Ada-Europe '96, number 1088 in Lecture Notes in Computer Science, pages 123--134. Springer-Verlag, June 1996.

[Min96] Naftaly Minsky. *Towards Alias-Free Pointers. Proc. of the 10th European Conference on Object Oriented Programming (ECOOP96)*, Linz, Austria July 1996.

[MNS95] Gail C. Murphy, David Notkin, Kevin Sullivan. "*Software Reflexion Models: Bridging the Gap between Source and High-Level Models*." ACM SIGSOFT 95 Symposium on the Foundations of Software Engineering, October 1995.

[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. *Proceedings of SIGSOFT'96: The Fourth Symposium on the Foundations of Software Engineering (FSE-4)*, San Francisco, CA, October 16-18, 1996.

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

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

[NVP98] James Noble, Jan Vitek, and John Potter. *Flexible alias protection*. Proc. 12th 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.

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