Language Support for Connector Abstractions

Jonathan Aldrich   Vibha Sazawal   Craig Chambers   David Notkin

Department of Computer Science and Engineering
University of Washington
Box 352350
Seattle, Washington, USA 98195-2350
+1 206 616-1846
`{jonal, vibha, chambers, notkin}@cs.washington.edu`

Abstract. Software connectors are increasingly recognized as an important consideration in the design and implementation of object-oriented software systems. Connectors can be used to communicate across a distributed system, coordinate the activities of several objects, or adapt one object's interface to the interface of another. Mainstream object-oriented languages, however, do not provide explicit support for connectors. As a result, connection code is intermingled with application code, making it difficult to understand, evolve, and reuse connection mechanisms.

In this paper, we add language support for user-defined connectors to the ArchJava language. Our design enables a wide range of connector abstractions, including caches, events, streams, and remote method calls. Developers can describe both the run-time semantics of connectors and the typechecking semantics. The connector abstraction supported by ArchJava cleanly separates reusable connection code from application logic, making the semantics of connections more explicit and allowing engineers to easily change the connection mechanisms used in a program. We evaluate the expressiveness and the engineering benefits of our design in a case study applying ArchJava to the PlantCare ubiquitous computing application.

1. Introduction

Software architecture is the high-level design of a system, composed of a set of components and the connections through which the components interact [GS93,PW92]. Object-oriented languages provide a natural object abstraction for components, and encourage developers to compose systems out of interacting objects. However, mainstream object-oriented languages do not provide explicit support for connections. Instead, connections are implicit in the object references in the heap, or are expressed indirectly using design patterns such as Proxy and Adaptor [GHJ+94].

Despite this lack of language support, connections are increasingly recognized as a crucial element of software systems. The software architecture literature has proposed a *connector* abstraction for connections, roughly corresponding to the class abstraction for components. In this context, a connector is a reusable design element that supports a particular style of component interactions. In a comprehensive taxonomy of connectors, Mehta et al. describe the wide variety of connectors used in

```
TaskListQuery q = new TaskListQuery();
q.list = "Water Plants";
sendMessage(taskServer,q,newClosure());
```

Figure 1. PlantCare code that sends a Rain message

software, including method calls, events, shared variables, adaptors, streams, semaphores, and many others [MMP00]. Connectors are particularly important in the context of distributed systems, where connector attributes such as bandwidth, synchronicity, security, reliability, and the wire protocol used may be crucial to the functionality and performance of the application.

Connector Libraries. Because of the lack of language abstractions for connectors, developers often implement connectors using library code. However, this implementation strategy often causes significant problems in the development and maintenance of software systems. Because connector abstractions do not exist, connector code is often mixed with component code, making both the component and the connector more difficult to reuse or evolve.

An improvement is to implement a connector as an Adaptor object that mediates between two components that may have different interface expectations. However, in this case the Adaptor implementation is tied to the interface of the components it connects; it cannot be re-used effectively to perform the same conceptual connection task between components that have a different interface. Similarly, when the Proxy pattern is used to implement a connector to an object that is present on a remote machine, a different Proxy implementation must be defined (or generated) for every distinct remote interface.

The PlantCare ubiquitous computing application, the subject of the case study described later in this paper, illustrates many of these problems. The PlantCare system is made up of sensors and robots that autonomously care for plants in a home or office environment [LBK+02]. This application presents a number of key research challenges for effectively building autonomous embedded systems.

PlantCare services communicate using Rain, a lightweight library for sending asynchronous XML messages over HTTP connections. Communication code that uses the Rain library is spread throughout the system, and its size and complexity often obscures the application logic of the system. As shown in Figure 1, sending even a very simple message in Rain is a multi-step process, and it can be quite tedious to write code for sending larger messages. It is difficult to identify the messages sent and received by PlantCare components because this information is spread throughout the code. Because of this scattering, it would be very difficult to change components to interact using a connector other than Rain. The situation might be somewhat improved by using the Proxy or Façade patterns to encapsulate communication code, but this improvement would come at the cost of greater code size. Despite these problems, we believe that PlantCare is ***not*** a poorly engineered, straw-man system; it is simply representative of the difficulty of writing well-modularized distributed applications using mainstream programming technology.

Tool Support. Communication infrastructures such as RMI [Jav97], CORBA [OMG95], and COM [Mic95] address these challenges by using tools to automatically generate proxies for communication with remote objects. These proxies encapsulate communication code, allowing application components to make remote method calls using the same syntax as local calls. Many CASE tools and code generation tools provide similar benefits. However, these infrastructures and tools fix a particular semantics for distributed communication—semantics based on synchronous method calls using particular encodings and wire protocols. While such tools may be ideal for applications that can accept the built-in semantics, they are inappropriate for applications that need different connector semantics. For example, the PlantCare developers decided that they needed to write a custom communication layer and wire protocol to support a very lightweight and adaptive form of communication appropriate to the ubiquitous computing domain. Although tools play an important role in implementing connectors, we believe that no single connection infrastructure will be sufficient for the diverse needs of all applications in the foreseeable future.

Our Approach. In this paper, we propose adding explicit language support for user-defined connectors. It is difficult to integrate user-defined connectors directly in a conventional object-oriented language such as Java, because connections between objects are not explicit in the source code, but are expressed implicitly through a pair of references. Instead, we present our design in the context of ArchJava, an extension to Java that allows developers to specify the software architecture of a system within the implementation. Because ArchJava already supports explicit connections between component objects, it can be easily extended to enable user-defined connectors to override the built-in connection semantics.

Our design allows developers to implement connectors using arbitrary Java code, supporting a very wide range of connector types. We evaluate the expressiveness of our design by implementing a representative subset of the connectors from Mehta et al.'s catalogue [MMP00]. A novel feature of our approach is that connectors define not just the run-time semantics of the connector, but also the typechecking strategy that should be used. As long as connector developers implement typechecking correctly for the domain of their connectors, our system provides a static guarantee of type safety to connector clients while still allowing connectors to link components with very different interfaces.

Our approach provides a clean separation of concerns. Each connector is modularly defined in its own class. Components interact with connectors in a clean way using Java's existing method call syntax. In our approach, the connector used to bind two components together is specified in a higher-level component, so that the communicating components are not aware and do not depend on the specific connector being used. Due to this design, it is easy to change the connectors in a system, while changing connectors may be very difficult in languages without explicit support for connector abstractions.

Organization. The rest of this paper is organized as follows. In the next section, we review the ArchJava language design through a simple peer-to-peer system example. Section 3 extends ArchJava with explicit support for connector abstractions, describing by example how they can be defined and used. We evaluate the expressiveness and the engineering benefits of our system in section 4, both by implementing a wide range of connectors and by applying ArchJava to part of the PlantCare ubiquitous computing application. We discuss related work in section 5 before concluding in section 6.

2. The ArchJava Language

ArchJava is a small extension to Java that allows programmers to express the software architecture of an application within the source code [ACN02a]. ArchJava's type system verifies *communication integrity*, the property that implementation code communicates only along connections declared in the architecture [MQR95,LV95,ACN02b]. This paper extends ArchJava by supporting much more flexible kinds of interactions along connections.

We illustrate the ArchJava language through PoemSwap, a simple peer-to-peer program for sharing poetry online. To allow programmers to describe software architecture, ArchJava adds new language constructs to support *components*, *connections*, and *ports*. The next subsection describes ArchJava's features for representing components and ports, while subsection 2.2 shows how developers can specify an architecture using components and connections. These sections review an earlier presentation of ArchJava [ACN02a].

2.1. Components and Ports

A *component* in ArchJava is a special kind of object that communicates with other components in a structured way. Components are instances of *component classes*, such as the `PoemPeer` component class in Figure 2. The `PoemPeer` component represents the network interface of the PoemSwap application.

Components in ArchJava communicate with each other through connected ports. A *port* represents a logical communication channel between a component and one or more components that it is connected to. For example, `PoemPeer` has a `search` port that provides search services to the PoemSwap user interface, and it has a `poems` port that it uses to access the local database of poems.

Ports declare two sets of methods, specified using the **`requires`** and **`provides`** keywords. A *provided* method is implemented by the component and is available to be called by other components connected to this port. For example, the search port provides searching and downloading methods that can be invoked from the user interface. Provided methods must be given definitions in the surrounding component class, as shown by the implementation of `downloadPoem` in Figure 2.

Conversely, each *required* method is provided by some other component connected to this port. In Figure 2, the `poems` port requires methods that get descriptions of all the poems in the database, retrieve a specific poem by its description, and add a poem to the database. A port may have both required and provided methods, but as shown in the example, it is common for a port to have only one or the other.

```
public component class PoemPeer {
  public port search {
    provides PoemDesc[] search(PoemDesc partialDesc) throws IOException;
    provides void downloadPoem(PoemDesc desc) throws IOException;
  }

  public port poems {
    requires PoemDesc[] getPoemDescs();
    requires Poem getPoem(PoemDesc desc);
    requires void addPoem(Poem poem);
  }

  public port interface client {
    requires client(InetAddress address) throws IOException;
    requires PoemDesc[] search(PoemDesc partialDesc, int hops, Nonce n);
    requires Poem download(PoemDesc desc);
  }

  public port interface server {
    provides PoemDesc[] search(PoemDesc partialDesc, int hops, Nonce n);
    provides Poem download(PoemDesc desc);
  }

  void downloadPoem(PoemDesc desc) throws IOException {
    client peer = new client(desc.getAddress());
    Poem newPoem = peer.download(desc);
    if (newPoem != null) {
      poems.addPoem(newPoem);
    }
  }
  // other method definitions...
}
```

Figure 2. The `PoemPeer` class represents the network interface of the PoemSwap application. `PoemPeer` communicates with other components through its ports. It provides a network search service to the rest of the application through the `search` port, and it accesses the poem database through the `poems` port. Finally, it communicates with other PoemSwap applications over a wide-area network using complimentary `client` and `server` ports.

A component can invoke a required method declared in one of its ports by sending a message to the port. For example, in Figure 2, after downloading a new poem from a peer, the `downloadPoem` method adds the new poem to the poem database with the call `poems.addPoem(newPoem)`. As this example shows, ports are concrete objects, and required methods can be invoked on ports using Java's standard method call syntax.

A *port interface* describes an interface used to communicate with multiple different components at run time. Port interfaces are to ports as classes are to objects. In fact, concrete port declarations such as `search` can be thought of as a convenient shorthand for a port interface and a field of that interface type. In the example, `PoemPeer` must communicate with many other PoemSwap peers through its `client` port interface, and it may serve requests from many peers through its `server` port interface. The two interfaces are symmetric, as each peer may act as both a client and a server.

The `client` port interface contains a connection constructor, named `client` after the surrounding port interface, that the `PoemPeer` can invoke in order to create a connection to a peer at the given `InetAddress`. `PoemPeer` instantiates a `client` port using this constructor in `downloadPoem` with the same **`new`** syntax used to create objects in Java. The `downloadPoem` method can then call the required method `download` on the newly created port.

The goal of ports is to specify both the services implemented by a component and the services a component needs to do its job. Required interfaces make dependencies explicit, reducing coupling between components and promoting understanding of components in isolation. For example, the `PoemPeer` component is implemented without any knowledge of what connection protocol will be used to connect it to its peers. `PoemPeer` expects a connector that has synchronous method call semantics, because the methods in the `client` port all return values, but any connector that conforms to this constraint can be used.

2.2. Software Architecture in ArchJava

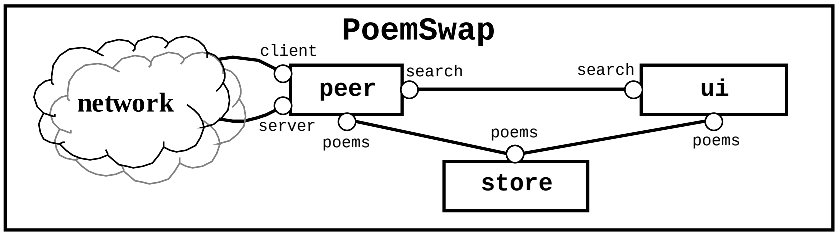
In ArchJava, hierarchical software architecture is expressed with *composite components*, which are made up of a number of subcomponents connected together. A *subcomponent*[1] is a component instance nested within another component. For example, Figure 3 shows how `PoemSwap`, the central component of the PoemSwap application, is composed of three subcomponents: a user interface, a poem database, and the peer discussed above. The subcomponents are declared as fields within `PoemSwap`.

In ArchJava, architects declare the set of permissible connections in the architecture using *connect patterns*. A connect pattern specifies two or more port types that may be connected together at run time. For example, the connect patterns in Figure 3 specify that both the user interface and the network interface connect to the `poems` port of the `PoemStore`, and that the `search` port of the user interface connects to the corresponding port of the network interface. The default typechecking rule for connect patterns ensures that for every method required by one or more of the connected ports, there is exactly one corresponding provided method with the same name and signature.

Actual connections are made using *connect expressions* that appear in the methods of a component. A connect expression specifies the concrete component instances to be connected in addition to the connected ports. In the example, the `PoemSwap` constructor makes three connections, one for each of the connect patterns declared in the architecture. A static check ensures that the types of the connected ports conform to the types declared in one of the connect patterns.

---

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



```
public component class PoemSwap {
  private final SwapUI ui = new SwapUI();
  private final PoemStore store = new PoemStore();
  private final PoemPeer peer = new PoemPeer();

  connect pattern SwapUI.poems, PoemStore.poems;
  connect pattern PoemPeer.poems, PoemStore.poems;
  connect pattern SwapUI.search, PoemPeer.search;

  public PoemSwap() {
    TCPConnector.registerObject(peer, “server”, POEM_PORT);

    connect(ui.poems, store.poems);
    connect(peer.poems, store.poems);
    connect(ui.search, peer.search);
  }

  connect pattern PoemPeer.client, PoemPeer.server {
    client(PoemPeer sender, InetAddress address) throws IOException {
      return connect(sender.client, PoemPeer.server);
    }
  };
}
```

Figure 3. A graphical and textual description of the PoemSwap architecture. The `PoemSwap` component class contains three subcomponents—a user interface, a poem store, and the network peer. Connect patterns show statically how these components may be connected, and the connect expressions in the constructor link the components together following these patterns. A final connect pattern shows how peers on different machines communicate, and includes a connection constructor that creates a connection when the `PoemPeer` requests one.

The built-in semantics of ArchJava connections binds required methods to provided methods, so that when a required method is called on one port, the corresponding provided method of the other port is invoked. For example, when the `PoemPeer` in Figure 2 invokes `addPoem` on its `poems` port, the invocation will be forwarded across the connection made in the PoemSwap architecture. The `addPoem` method implementation provided by the `poems` port of the `PoemStore` (not shown) will be invoked.

Connection Constructors. Each connect pattern must provide a connection constructor for each of the required connection constructors declared in the connected ports. A connection constructor is named after the port that required the constructor, and the first argument is the component that requested the connection. The other arguments match the ones declared in the corresponding connection constructor. For

```
connect pattern PoemPeer.client, PoemPeer.server with TCPConnector {
    client(PoemPeer sender, InetAddress address) throws IOException {
      return connect(sender.client, PoemPeer.server)
        with new TCPConnector(connection, address, POEM_PORT);
    }
  };
```

Figure 4. The final connect pattern in PoemSwap, augmented with connector specifications. The connect pattern specifies that `TCPConnector` should be used to typecheck the connection statically. The connect expression instantiates a `TCPConnector` object to connect to the remote peer, passing to the constructor a reification of the connection together with the address of the remote peer.

example, the `client` port in component class `PoemPeer` requires a connection constructor that accepts an `InetAddress`. Since `PoemPeer.client` is one of the ports in this connect pattern, the pattern declares a connection constructor with two arguments—the `PoemPeer` that requested the connection and an `InetAddress`. The body of a connection constructor must return a connection that matches the surrounding connect pattern. One of the connected ports must be the appropriate port of the component that requested the connection (`sender.client` in the example).

3. Connector Abstractions in ArchJava

In this section, we describe the new language features and libraries that support connector abstractions in ArchJava. We extend the syntax of connect patterns and connect expressions to describe which connector abstractions should be used to typecheck and implement the connections. Subsection 3.1 demonstrates these language features by examples, showing how a user-defined TCP/IP connector can be used to connect different `PoemSwap` peers across a wide-area network. New connectors can be written using the `archjava.reflect` library, described in Subsection 3.2, which reifies connections and required method invocations. Finally, subsection 3.3 shows how the TCP/IP connector can be implemented using this library.

3.1. Using Connector Abstractions

Connector Typechecking. Instead of using ArchJava's default typechecking rules, connect patterns can specify that a user-defined connector should be used for typechecking instead. For example, Figure 4 shows the syntax missing from the connect pattern at the end of Figure 3. After declaring the component ports to be connected, the connect pattern can specify a user-defined connector class to be used for typechecking using the syntax **`with`** <*connector class*>. The connector used can be any subclass of `archjava.reflect.Connector`, which defines a `typecheck` function that can be overridden by subclasses. In the example, when the `PoemSwap` component class is compiled, the compiler loads the

`TCPConnector` class, creates an instance, and invokes the `typecheck` method on the `TCPConnector` to check the validity of the connect pattern. This typechecking replaces the default ArchJava typechecking semantics, allowing the connector abstraction to define arbitrary typechecking rules.

In the case of `TCPConnector`, the `typecheck` method first invokes the standard ArchJava typechecker, and then additionally checks that all arguments and results of all methods in the connection are subtypes of the `Serializable` interface. Because the `TCPConnector` uses Java's serialization mechanism to send method arguments and results across a network, a run-time error will result if the method arguments and results are not serializable. By defining its own typechecking semantics to extend those of ArchJava, the `TCPConnector` can detect this error at compile-time[2].

Instantiating Connectors. Connectors are instantiated whenever a connect expression is executed at run time. Connect expressions can use the syntax **`with`** *<expression>* to specify the connector instance that should be used for the connection. The expression in the **`with`** clause must be have a type that is a subclass of the connector type declared in the corresponding connect pattern, to ensure that the connector implementation used at run time matches the connector that was used to typecheck the connection statically. In the example, the expression creates a new `TCPConnector` object, passing the address and the TCP/IP port of the remote peer to the constructor of the connector. When the connect expression is executed at run time, the **`with`** clause will be evaluated, creating a `TCPConnector` object to be used in the connection.

Inside the **`with`** clause of a connect expression, the connection being created is reified in the `connection` variable, which has type `archjava.reflect.Connection`. This object contains the types of the components and ports in the connection, as well as the component instances being connected. Connectors need this information in order to implement their functionality, and so we pass the connection object to the constructor of `TCPConnector` along with the address and TCP/IP port.

In the case of `PoemSwap`, the component to be connected to the sender is a peer on a remote machine, and so we cannot use a direct reference to it in the connect expression. Instead, we specify the type of the remote component so that it will match the surrounding connect pattern, and the `TCPConnector` uses the address passed to the constructor to communicate with the remote component.

3.2. The `archjava.reflect` Library

Connector abstractions are defined using the `archjava.reflect` library, whose most important classes and methods are shown in Figure 5. This library defines a

---

[2] This check would have been handy when testing the PoemSwap application. Before customized typechecking was implemented, we got run time errors because we forgot to make poems serializable.

```
public class Connector {
  public Connector(Connection c);
  protected Connector(Object components[], String portNames[]);

  public final Connection getConnection();

  public Error[] typecheck();
  public Object invoke(Call c) throws Throwable;
}

public final class Connection {
  public Port[] getPorts()
  public Connector getConnector()
}

public final class Port {
  public String getName();
  public Method[] getRequiredMethods();
  public Method[] getProvidedMethods();
  public Object getEnclosingObject();
}

public final class Method {
  public String getName();
  public Type[] getParameterTypes();
  public Object invoke(Object args[]) throws Throwable;
}

public final class Type {
  public String getName();
}

public final class Call {
  public Method getInvokedMethod();
  public Object[] getArguments();
}
```

Figure 5. The `archjava.reflect` library includes classes reifying connectors, connections, ports, methods, types, and calls. User-defined connector classes extend the `Connector` class, overriding the `invoke` or `typecheck` methods to define customized dynamic and/or static semantics, respectively.

`Connector` class that user-defined connector classes extend, as well as classes that reify connections, ports, and methods.

Class `Connector` provides a hook for defining customized connectors. Connector abstractions can define custom typechecking semantics by overriding the `typecheck` method, which is called at compile time to typecheck a connect pattern, returning a possibly empty array of errors. Run-time connection behavior can be defined by overriding the `invoke` method, which accepts a `Call` object reifying an invocation on a required method. The default implementation finds the corresponding provided method and invokes it, passing the resulting return value or exception back to the caller.

`Connector` provides a public constructor that accepts a reified connection object. A second constructor creates its own connection object from the specified arrays of components and corresponding port names. This constructor allows connections to be created without being declared in a connect pattern, so it is accessible only to `Connector` subclasses.

Classes `Connection`, `Port`, `Method`, `Type`, and `Call` reify the connection that is associated with the connector, along with its ports and method signatures. Unlike `Connector`, user-defined connectors do not extend these classes, but instead may use them as a library for getting information about the current connection. This information, accessible through the `getConnection` method of `Connector`, can be used statically when typechecking or dynamically when dispatching a required method invocation. For example, the connector can invoke provided methods by calling `invoke` on the relevant `Method` object.

3.3. Implementing Connector Abstractions

`TCPConnector` is designed to create a TCP/IP network connection between the local host and the remote host and TCP/IP port specified in the constructor. When a method is called on the connection, the `TCPConnector` serializes the arguments of the method and ships them across the TCP/IP network connection. The constructor of the `PoemSwap` component registers its `PoemPeer` with the `TCPConnector`, which starts a daemon listening at the appropriate TCP/IP port. The daemon will read the method's name and arguments, and invoke the appropriate method from the connected port. The process is reversed in order to send the method's return value back over the network.

Figure 6 shows how the run-time semantics of `TCPConnector` can be defined in Java code. The example shows primarily the interface of the connector and how it uses the `archjava.reflect` library. We omit the network, serialization, and threading code in the `TCPDaemon` helper class.

The `TCPConnector` constructor passes the connection object on to the `Connector` superclass, and then creates a `TCPDaemon` object to handle the TCP/IP network connection. `TCPConnector` overrides `invoke` by determining which required method was called, and passing the name of the method, its parameter types, and the actual call arguments to the `TCPDaemon`. The `TCPDaemon` sends this data over the TCP/IP network connection.

At the other side of the network, `PoemSwap` has used `registerObject` to register a `PoemPeer` object. The `registerObject` method starts a `TCPDaemon` listening at the assigned port. When the daemon receives an incoming connection, it creates a `TCPConnector` object to represent that endpoint of the connector. The daemon uses the non-public `TCPConnector` constructor, passing the object to be connected and the name of its connected port to the constructor. Since the originating connection was created on the other machine, the constructor does not have a local `Connection` object to pass to the ordinary superclass constructor, so it calls a constructor that takes two arrays of objects and port names, respectively, and generates a local `Connection` object based on this information.

```java
public class TCPConnector extends Connector {
  // data members
  private TCPDaemon daemon;

  // public interface
  public TCPConnector(Connection conn, InetAddress host, int prt)
                                            throws IOException {
    super(conn);

    daemon = TCPDaemon.getDaemon(host, prt, this);
  }

  public Object invoke(Call call) throws Throwable {
    Method meth = call.getInvokedMethod();
    return daemon.sendMethod(meth.getName(), meth.getParameterTypes(),
                             call.getArguments());
  }

  public static void registerObject(Object o, String portName, int prt)
                                    throws IOException {
    TCPDaemon.getDaemon(prt).register(o, portName);
  }

  // interface used by TCPDaemon
  TCPConnector(TCPDaemon daemon, Object receiver, String portName) {
    super(new Object[] { receiver }, new String[] { portName });
    this.daemon = daemon;
  }

  Object invokeLocalMethod(String name, AType parameterTypes[],
                                Object arguments[]) throws Throwable {
    // find method with parameters that match parameterTypes
    Method meth = findMethod(name, parameterTypes);
    return meth.invoke(arguments);
  }

  // typecheck defined in Figure 7
}
```

Figure 6. The `TCPConnector` class extends the `archjava.reflect.Connector` class to define the dynamic semantics of a connector based on a TCP/IP network connection. The `invoke` method passes the method name, parameter types, and arguments to a daemon that uses Java's serialization facilities to send them over a TCP/IP network connection. The daemon at the other end of the connection, created when the other peer called `registerObject`, calls `invokeLocalMethod` on a `TCPConnector` object, which identifies the right method to call and invokes it.

When the daemon receives an incoming method, it finds the `TCPConnector` associated with the receiver object and calls `invokeLocalMethod`. `invokeLocalMethod` uses the `findMethod` helper function to identify the provided method that matches the method name and parameter types, then invokes the method through a reflective call. The result, or any exception that is thrown, will be packaged back up by the `TCPDaemon`, sent back over the network, returned to the implementation of `invoke` in the source `TCPConnector`, and returned to the caller from there.

```
// extension of the code in Figure 6
public class TCPConnector extends Connector {
  public TCPConnector(Connection c) {
    super(c);
    // this constructor may only be used for compile time typechecking
    if (!c.isCompileTime()) {
      throw new RuntimeException("Must pass a remote port at run time");
    }
  }

  public Error[] typecheck() {
    // First invoke Java’s typechecking
    Error [] errors = super.typecheck();
    if (errors.length > 0)
      return errors;

    // ensure that there are exactly two connected ports
    Connection c = getConnection();
    if (c.getPorts().length != 2) {
      return new Error[] {
        new Error("TCPConnectors must connect 2 ports”, c);
      };
    }

    // ensure all arguments and results are Serializable
    ...
  }
}
```

Figure 7. Overriding `typecheck` in the `TCPConnector` class to ensure that TCP connectors only connect two ports.

User-Defined Typechecking. Figure 7 shows the definition of the `typecheck` method of `TCPConnector`. This method, inherited from `archjava.reflect.Connector`, is called at compile time for each connect pattern in the system that uses a `TCPConnector` for typechecking.

Connectors that define custom typechecking must implement a constructor that takes a single argument of type `archjava.reflect.Connection`. This object is initialized by the compiler to hold the list of ports in the connect pattern, and for each port, the types of its parameters and return value. The `TCPConnector` ensures that this constructor is only used at compile time by throwing an exception if the `Connection` object reports that it is a run-time connection.

Connectors define their typechecking semantics by overriding the `typecheck` method from `TCPConnector`. This method returns a possibly empty array of `Error` objects describing any semantic errors in the connect pattern. The `Error` class encapsulates a `String` describing the problem as well as a syntax element (a `Connection`, `Port`, or `Method`) that describes where the error occurred, allowing the compiler to give the reported error an accurate line number.

The `TCPConnector` begins by running the standard `typecheck` method defined in its superclass. It returns any errors found by this method. If standard typechecking succeeds, the `TCPConnector` next ensures that there are only two ports in the connection, because `TCPConnector` is not designed to handle the multi-way connections that are allowed by the ArchJava syntax. Finally, code (not shown) visits every required and provided method in the connection, making sure that all method arguments and results are either primitive types or are `Serializable`, so that the `TCPDaemon` will be able to serialize them successfully at run time.

4. Evaluation

We have implemented language support for connector abstractions in the ArchJava compiler, available for download at the ArchJava web site [Arc02]. Thus, all examples in this paper, including PoemSwap and PlantCare, are simplified versions of live code.

We evaluate our design in two ways. In the next subsection, we evaluate the expressiveness of our connector abstraction mechanism by describing how a wide range of connectors can be implemented. In the following subsection, we evaluate the engineering benefits of connector abstractions with a case study on the PlantCare ubiquitous computing application. Subsection 4.3 analyzes the case study and reports feedback from the developers of PlantCare.

4.1. Expressiveness

In order to evaluate the expressiveness of our connector abstraction mechanisms, we use Mehta et al.'s taxonomy of connectors as a benchmark for our design [MMP00]. The taxonomy describes eight major types of connectors: procedure call, event, data access, linkage, stream, arbitrator, adaptor, and distributor connectors. We discuss each connector type in turn, describing which species of that connector can benefit from using connector abstractions, and giving examples of their definition or use where appropriate.

Connector Implementation. Connectors can be implemented in a wide variety of ways, each with its own benefits and drawbacks. For example, in addition to our connector abstractions, connectors could be built into the language, expressed idiomatically through a design pattern, or described using ArchJava's component construct.

The key benefit of using connector abstractions is that the same connector can be reused to support the same interaction semantics across many different interfaces, while still providing a strong, static guarantee of type safety to clients. For example, the `TCPConnector` is able to connect any two compatible ports, as long as the arguments to methods in those ports are `Serializable`. Other solutions that guarantee type safety require separate stub and skeleton code to be written for each interface, causing considerable code duplication and hindering reuse and evolution. Alternatively, a generic library interface for sending objects across a TCP/IP connection could be used, but this solution does not guarantee that the messages sent and received across the connection have compatible types, so run-time errors are possible.

The main drawback of using connector abstractions is that they are defined using a reflective mechanism. Although connectors can define typechecking rules for their clients, there is no way to statically check that a connector's implementation performs the communication in a type-safe way. Also, there is some run-time overhead associated with reifying a method call so that a connector can process it dynamically. Thus, in situations where a connector is not reused across different interfaces, it may be better to use objects or components to implement the connector.

We now go through the eight connector types identified by Mehta et al., analyzing the situations in which the use of connector abstractions might be beneficial. All of the connector abstraction examples described here are available for download as part of the ArchJava distribution [Arc02].

Procedure Call. Procedure call connectors enable the transfer of control and data through various forms of invocation. Although most programming languages provide explicit support for procedure calls, there are a number of semantic issues that justify user-defined procedure call connectors. For example, parameters could be passed by reference, by value, by (deep) copy, etc.; calls could be synchronous or asynchronous; calls could use one to many broadcast semantics, many-to-one collecting semantics, or conceivably even many-to-many semantics.

ArchJava's connector abstractions are well-suited to implementing procedure call connectors, because of the procedure-call syntax used in ports and because the mechanism allows connectors to define many different kinds of method-call semantics, which are then reusable across different port interfaces.

As an example, we have implemented an `AsynchronousConnector` that accepts incoming required method calls, returns to the sender immediately, and then invokes the corresponding provided method asynchronously in another thread. We have also implemented a `SummingBroadcastConnector` that accepts a incoming method call, broadcasts it to all connected components, and sums the results of all the invocations before returning the sum to the original caller. This second connector relies on ArchJava's n-way connections, which can connect more than two ports. Both connectors implement appropriate typechecking; for example, the `AsynchronousConnector` ensures that all methods in connected ports return **`void`**, while the `SummingBroadcastConnector` ensures that all of the methods return a number. The `TCPConnector` described above is a procedure call connector that connects components running on different virtual machines.

Event. Event connectors support the transfer of data and control using an implicit mechanism, where the producer and consumer of an event are not aware of each other's identity. Semantic issues with event connectors include the cardinality of producers and consumers, event priority, synchronicity, and the event notification mechanism.

Events are often implemented as inner-class callbacks in languages such as Java, but this technique can make programs very difficult to reason about and evolve, as it is hard to see which components might be communicating through an event channel. ArchJava's explicit support for connections makes it easy to use events for communication between loosely coupled components, while showing the connection in the program's software architecture to aid in program understanding and evolution. Connector abstractions provide additional benefit by allowing components to communicate using different event semantics. For example, we have implemented a `EventDispatchConnector` that enqueues event notifications and dispatches them to consumers based on their priority.

The PlantCare application, described below in subsection 4.2, uses a user-defined connector to support asynchronous event-based communication across a loosely coupled ad-hoc network.

Data Access. Data access connectors are used to access a data store, such as a SQL database, the file system, or a repository such as the Windows registry. Issues in data access components include initialization and cleanup of connections to data sources, and the conversion and presentation of data. Conventional library-based techniques are appropriate for implementing many kinds of data access connectors. However, connector abstractions can be used to provide a convenient view of the data source, or adding semantic value to a data source in a reusable way. For example, one could imagine a connector that provides an object-oriented view of a relational database, translating each row of each table into an object and providing a collection-like access to clients. We have implemented a `CachingConnector`, which caches the results of method calls to a data store and returns the same result if the method is called again with identical arguments.

Linkage. Linkage connectors bind a name in one module to the implementation provided by another module. Examples of linkage connectors include imported names and references to names defined in other source files. ArchJava's connector abstractions are intended to connect object instances at run time, not link names at compile time. Therefore, Linkage connectors are outside of the scope of ArchJava's connector abstraction design.

Stream. Stream connectors support the exchange of a sequence of data between loosely-coupled producer and consumer components. Semantic issues with streams include buffering, bounding, synchronicity, data types, data conversion, and the cardinality of the producers and consumers. Many of these issues can be encapsulated within a reusable connector abstraction. For example, we have developed a `BufferedStreamConnector` that implements a stream with a bounded buffer size, supporting one producer but an arbitrary number of consumers. The `BufferedStreamConnector` is reusable for streams of many different data types, but checks that the types of data produced and consumed match.

Arbitrator. Arbitrator connectors provide services that coordinate and facilitate interactions among components. Examples of arbitrators include semaphores, locks, transactions, fault handling connectors with failover, and load balancing connectors. Semaphores and locks typically have the same interface no matter which components they connect, and so they are probably best implemented using ordinary objects or as ArchJava components. However, more sophisticated arbitrators can benefit from ArchJava's connector abstraction mechanism. For example, we have built a

`LoadBalancingConnector` that accepts incoming method calls from a client and distributes them to a bank of server components based on the current server loads. The `LoadBalancingConnector` is reusable across any client interface, while still providing typechecking between clients and services. We have also implemented a barrier synchronization connector. Components invoke a different method on the barrier after each stage of work, and the barrier ensures that all its clients have called a given barrier method before it allows any of the method calls to return.

Adaptor. Adaptor components retrofit components with different interfaces so that they can interact. Adaptors may convert data formats, adapt to different invocation mechanisms, transform protocols, or even make presentation changes like internationalization conversions. Well-known design patterns such as Adaptor, Wrapper, and Façade are often used to implement adaptors. However, connector abstractions can be useful for performing similar adaptations to different interfaces. For example, the `RainConnector` in section 4.2 below adapts data types using structural subtyping, so that two components can communicate with different datatypes as long as the data sent in a message has all of the information expected by the receiver.

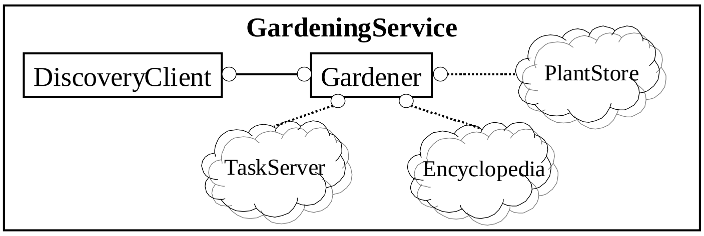
Distributor. Distributor connectors identify paths between components and routes communication along those paths. Distributors are not first-class connectors, but provide routing services to other connectors. Both the `EventDispatchConnector` described above and the `RainConnector` described below include distributor functionality.

Summary. As the discussion above makes clear, ArchJava's connector abstractions are very flexible, supporting a very wide range of different connector types. Some kinds of connectors are most clearly expressed using conventional mechanisms such as objects and components. However, connector abstractions provide a unique level of reusability across port interfaces while still providing clients with a strong static guarantee of type safety.

4.2. PlantCare Case Study

In order to evaluate the engineering benefits of user-defined connector abstractions, we performed a case study with the PlantCare ubiquitous computing application [LBK+02]. PlantCare is a project at Intel Research Seattle that uses a collection of sensors and a robot to care for houseplants autonomously in a home or office environment. This application illustrates many of the challenges of ubiquitous computing systems: it must be able to configure itself and react robustly to failures and changes in its environment.

The Gardening Service. Figure 8 shows the architecture of the gardening service, one of several services in the PlantCare system. The gardening service consists of a central gardener component that uses three external services as well as a client for a well-known discovery service. The gardener periodically executes a cycle of code



```
public component class GardeningService {
  private final Gardener gardener = new Gardener(getServiceID());
  private final DiscoveryClient client = new DiscoveryClient();

  connect client.discovery, gardener.discovery;

  connect pattern Gardener.PlantInfo, PlantStore.PlantInfo
    with RainConnector {
      PlantInfo(Gardener sender, ServiceID id) {
        return connect(sender.PlantInfo, PlantStore.PlantInfo)
                 with new RainConnector(connection, id);
      }
    }
  // other architectural connections not shown
}

public component class Gardener extends StateMachineNode {
  public port discovery {
    requires ServiceID find(String serviceType);
  }
  public port interface PlantInfo {
    requires PlantInfo(ServiceID id);
    requires void statusQuery();
    provides void statusReply(PlantStatus data);
  }
  private PlantInfo plantInfoPort;

  public startStateCycle() {
    ServiceID ID = discovery.find(“Plant Store”);
    plantInfoPort = new PlantInfo(ID);
    plantInfoPort.statusQuery();
    ...
  // remaining Gardener implementation not shown
}
```

Figure 8. The architecture of the PlantCare gardening service

that cares for plants as follows. First, the gardener requests from the `PlantStore` a list of all the plants in the system and the sensor readings from each plant. For each plant, it queries the `Encyclopedia` to determine how that plant should be cared for. After comparing the recommended and actual plant humidity levels, it adds or removes watering tasks from the `TaskServer` so that each plant remains in good health.

We have chosen to include the interfaces of relevant external services as part of the gardening service architecture, because then we can use the connectors in the architecture to reason about the protocols used to communicate with these services. A more conventional architectural depiction would represent these protocols as connectors in an enclosing architecture. However, in ubiquitous computing systems, there is no way to statically specify the entire enclosing architecture, because the services available in a system may change frequently as devices move and connections fail. Instead, the gardening service architecture includes a partial view of the surrounding architecture, including the external components with which the gardener communicates.

Below the visual architectural diagram in Figure 8 is the ArchJava code describing the architecture. The concrete `Gardener` and `DiscoveryClient` component instances are declared with final fields. The connect declaration linking the `discovery` ports of the `client` and the `gardener` is syntactic sugar for a connect pattern and a corresponding connect expression.

The connect pattern links the `PlantInfo` ports of the gardener and the plant store. When the gardener requests a new connection, the provided connection constructor specifies that it should be connected with a `RainConnector`, using a `ServiceID` to identify the location of the remote `PlantStore` component. The other connect patterns, although omitted from this diagram, are similar.

The `RainConnector` class implements the Rain communication protocol used in the PlantCare system. When methods are invoked through connections of type `RainConnector`, the user-defined connector code will package the method name and arguments as an XML message, send them over a HTTP connection, and call the appropriate provided method on the other side. Since Rain messages are asynchronous and do not return a response, `RainConnector` also defines a custom typechecker that verifies that methods in the connected ports have a **`void`** return type. Although `RainConnector` is similar to `TCPConnector` in that both connect components that may be located on different hosts, it provides very different semantics (asynchronous messages vs. synchronous method calls), demonstrating the versatility of ArchJava's connector abstractions.

The `RainConnector` implementation is similar to the `TCPConnector` defined earlier. The connector uses the name of the method called as the name of the XML message to be sent. The method arguments are serialized and sent over the network using the same Rain library that is currently used by the PlantCare application. Because Rain messages are asynchronous, the `RainConnector` returns immediately after sending a message, without waiting for an acknowledgement or response.

The `Gardener` class has a concrete port for discovery, but port interfaces for communicating with other components. This is a natural choice, because discovery is a fundamental service that must be in place in order for the `Gardener` to dynamically discover other available services. The discovery interface allows the `Gardener` to look up a service by its type. It returns a `ServiceID` data structure that can then be used in a connection constructor to connect to other components.

The code in `startStateCycle` shows the beginning of the cycle of code that the `Gardener` executes when caring for plants. The code uses the discovery service to find the `ServiceID` of an available `PlantStore` service. It then allocates a new `PlantInfo` port and stores it in a variable. The final line of code shown sends an asynchronous message through the newly allocated port, querying the status of the plants in the system. The `PlantStore` will reply with another asynchronous message, which will be translated by the `RainConnector` into a call to the `statusReply` method, which carries out the next stage in the cycle. If an internal timer expires before the `statusReply` message is received, the gardener assumes that the `PlantStore` component (or an intervening network link) has failed, and restarts the state cycle, using the discovery service once again to connect to a functioning `PlantStore`.

4.3. Discussion

In this section, we analyze the results of our case study according to a number of criteria: program adaptivity, program understanding, program correctness, and software evolution. Finally, we report feedback from the developers of the PlantCare application.

Adaptivity. Ubiquitous computing services must be robust to communication failures and to failures in other services. The gardening service uses a simple strategy for handling failure: if it does not receive a response to a query within a timeout interval, it begins the state cycle again, re-establishing connections to the components it depends on. ArchJava supports this adaptive strategy by providing customizable connectors that can be created and configured at run time.

Program Understanding. The ArchJava version of the gardening service code has a number of characteristics that make it easier to understand the service's implementation. In the Java version, the information about which messages are sent and received is spread throughout the source code. Figure 8 shows how the ArchJava architecture documents the sent and received messages explicitly as required and provided methods in the ports of `Gardener`, making it easier to understand the interactions between the gardener and other services.

Figure 8 also shows how the ArchJava source code documents the architecture of the service, showing which other services the gardener depends on. This information is obscured in the original gardener source code; it would have to be deduced from the types of messages exchanged. Another benefit is that the connector specification explicitly documents that the Rain communication protocol is used between components. This would be especially valuable if the gardener used different protocols to communicate with different external services, as may often be the case in heterogeneous ubiquitous computing systems.

Java Version:

```
protected void handleMessageIn(Message m) {
  if ... { ... // cases for plant status messages above...
  } else if (msg instanceof PlantInfoReply) {
    // case for plant info message
    PlantInfoReply p = (PlantInfoReply) msg;
    careMap.put(p.name,p);
    state = AWAITING_TASKS;
    sendTasksRequest();
    return;
  } else if (msg instanceof TaskListReply) {
    // case for task reply message below...
}

protected void sendTasksRequest() {
  try {
    TaskListQuery q = new TaskListQuery();
    q.list = "Water Plants";
    sendMessage(taskServer,q,newClosure());
  } catch (Exception ex) {
    // an error occurred, restart the cycle
    ex.printStackTrace();
    resetState();
  }
}
```

ArchJava Version:

```
void infoReply(PlantInfoReply data) {
  careMap.put(data.name, data);
  state = AWAITING_TASKS;
  try {
    task.taskQuery("Water Plants");
  } catch (Exception ex) {
    // an error occurred, restart the cycle
    ex.printStackTrace();
    resetState();
  }
}
```

Figure 9. A comparison of the old and new versions of the `Gardener` code that responds to the `PlantInfoReply` message.

Figure 9 compares the Java and ArchJava versions of the code that responds to a `PlantInfoReply` message from the encyclopedia. Here, ArchJava's abstraction mechanisms for inter-component communication make the application logic of the gardener clearer. In the original Java code, a single `handleMessageIn` method responds to all incoming messages. The `PlantInfoReply` message is one case in a long list of messages; the code stores the plant care information in an internal data structure and then calls a separate `sendTasksRequest` function to send out the next batch of messages. In the ArchJava version, this response code is more cleanly encapsulated in a single method, which responds to the original message and then sends the next set of messages through the task port. The process of sending a message is also simpler and cleaner in ArchJava. The programmer simply calls a method in the `taskPort`, rather than constructing a custom message and sending it using the Rain library.

Correctness. The ArchJava language performs a number of checks that help to ensure the correctness of the `GardenerService` implementation. For example, the `RainConnector` typechecker verifies interface compatibility between the ports of `Gardener` and the connected ports of the external services at compile time. In the original Java code, this problem would show up as a run time error when a component does not recognize a message that was sent to it.

ArchJava also verifies communication integrity [MQR95,LV95,ACN02b], a property which guarantees that the `Gardener` only communicates with the services declared in the `GardeningService` architecture (We assume that the gardener does not directly use Java's networking library, a property that could also be checked in a straightforward way). This property guarantees that the architecture can be relied on as an accurate representation of the communication in the system, increasing the program understanding benefits of architecture.

Software Evolution. Because of ArchJava's explicit abstractions for ports and connectors, some evolutionary steps are easier to perform. For example, if a service needs to interact with a device that cannot generate XML messages, we can replace `RainConnector` with a new connector type that can communicate with the more restricted device. Also, we can reuse an existing service in a new environment by simply inserting adaptor components or connectors that retrofit the old service to the message protocol expected by the new environment. In both cases, ArchJava's explicit descriptions of component interfaces and connections make architectural evolution easier.

An important criterion to consider in the evolvability of a system is the degree to which the system's modularization hides information within a single module. One benefit of the ArchJava version of the gardening service is that the gardener's functionality is encapsulated in `Gardener` while the communication protocol used is encapsulated in `GardeningService`. The ports of `Gardener` serve as the interfaces used to hide this information. Thus, in the ArchJava code, the gardening functionality can be changed independently of the communication protocol, facilitating evolution of this service.

Developer Feedback. Perhaps the most important evaluation criterion is feedback from the developers of PlantCare. We found that the developers were able to understand the ArchJava notation fairly quickly. They said that the `GardeningService` architecture captured their informal architectural view of the system well. Finally, they agreed that ArchJava was able to provide the benefits describe in the analysis above. We are currently working with them to put ArchJava to production use in a future ubiquitous computing system.

5. Related Work

Software Architecture. Connectors have been studied most extensively in the context of the software architecture literature. Mehta et al. provide a thorough classification of software connectors [MMP00]. They categorize connectors based on the services they provide for *communication*, *coordination*, *conversion*, and *facilitation*. Their study also identifies many common types of connectors, and the dimensions of variability within each connector type. In section 4, we use this study as a benchmark for our design, choosing representative connectors from each major category, and evaluating how well our design supports the implementation of these connectors.

Most architecture description languages (ADLs) support the specification or implementation of software connectors [MT00]. For example, Wright specifies the temporal relationship of events on a connector and provides tools for checking properties such as freedom from deadlock [AG97]. SADL formalizes connectors in terms of theories and describes how abstract connectors in a design can be iteratively refined into concrete connectors in an implementation [MQR95]. Rapide specifies connectors within a reactive system using event traces [LV95].

Several ADLs provide tools that can generate executable code from an architectural description. UniCon's tools use an architectural specification to generate connector code that links components together [SDK+95]. C2 provides runtime libraries in C++ and Java that implement C2 connectors [MOR+96]. Darwin provides infrastructure support for implementing distributed systems specified in the Darwin ADL [MK96]. These code generation tools, however, support a limited number of built-in connector types, and developers cannot easily define connectors with custom semantics.

User-Defined Connectors. The work most similar to our own is a specification of how user-defined connector types can be added to the UniCon compiler [SDZ96]. Their design is fairly heavyweight, as connector developers must understand the details of several phases of the compiler. However, it allows new connectors to be tightly integrated into the compiler system, potentially allowing new kinds of architectural analysis to be defined over these connectors. In contrast, ArchJava's connector abstractions are lightweight, and a wide range of connectors can be implemented with knowledge of a small library interface.

Dashofy et al. describe how off-the-shelf middleware can be used to implement C2 connectors [DMT99]. Their work differs from ours in that the semantics of the connectors is fixed by the C2 architectural style, while our connector abstractions are intended to support a wide range of architectural styles.

Mezini and Ostermann describe language support for adaptor connections that allow components with different data models to work together [MO02]. Their language makes wrapper code less tedious to write, and provides support for the difficult problem of maintaining consistent wrapper identity. ArchJava's connector abstractions provide weaker support for adaptor connections, but facilitate a range of connector types beyond just adaptors.

Object-Oriented Languages. A number of proposals have added connection support to object-oriented languages such as Java. For example, ComponentJ [SC00] and ACOEL [Sre02] as well as the original design of ArchJava [ACN02a] all provide primitives for linking components together with connections. However, these languages all fix the semantics of connections to the same synchronous method call semantics used by Java.

Many languages have also provided support for distributed applications in the context of an object-oriented programming language. Perhaps the first such system was Emerald, which provides a pure object model with support for remote method call and mobile objects within a local area network [JLH+88]. More recent systems include Lana, which provides an asynchronous method call primitive for designing autonomous distributed systems [BRP02]. The Cells project builds a distributed system out of Cell modules with language-level support for remote method calls, mobile objects, and a security model [RS02]. Each one of these systems defines its own semantics for network communication, rather than defining a connector abstraction mechanism that can encompass a broad range of connector semantics.

Hydro is a language under development at the University of Washington that explicitly supports ubiquitous computing applications [LC02]. Because ubiquitous computing is an interesting and challenging domain, we use a ubiquitous computing application as a case study in this paper. Hydro's support for ubiquitous computing goes beyond distributed connectors to include a semi-structured data model. In contrast, ArchJava provides a connector abstraction that can be used in ubiquitous computing applications and in many other domains as well.

Aspect-Oriented Programming. Aspect-oriented programming (AOP) languages allow programmers to separate code that implements different application concerns. Hannemann and Kiczales showed how the AspectJ language can be used [HK02] to implement distribution in a classroom support application. Aspect-oriented programming developed out of meta-object protocols, which allow programmers to define how an object should react to events like method calls [KRB91]. The composition filters approach to AOP allows developers to interpose filter objects that can inspect incoming method calls and perform operations like translation, adaptation, and forwarding on the messages [BA01]. ArchJava's connector abstractions are similar to composition filters, but instead of processing all messages called on a single object, they process messages exchanged between two component objects in an architecture.

Distributed System Infrastructures. A number of libraries and tools have been defined to support distributed programming. Commercial examples include RPC as well as COM [Mic95], CORBA [OMG95], and RMI [Jav97]. These systems offer a convenient method-call interface for remote communication, much like the interface provided by ArchJava's connector abstractions. Furthermore, these systems check statically that communication through their connections is well-typed. Infrastructures support some flexibility—for example, RMI allows the developer to specify the wire protocol to be used, and CORBA provides an event service that can be used in place of remote method calls. However, each of these systems defines a particular semantics (usually synchronous method call) for the connections it supports, rather than providing a general interface that programmers can implement in various ways to support their application-specific needs.

CASE Tools. Several computer-aided software engineering tools, including Consystant and Rational Rose RealTime, generate code to connect components together. This connection code can range from stubs and skeletons for an infrastructure like CORBA or RMI to wires that connect different processors in an embedded system. Like many of the technologies discussed above, these tools typically support a fixed set of connectors.

6. Conclusion

This paper described a technique for adding explicit support for connector abstractions to the ArchJava programming language. In our system, connector abstractions can be defined using a very flexible reflective library-based mechanism. We have evaluated the expressiveness of our technique by implementing representative connectors from a wide range of connector types, and we have evaluated the engineering consequences in a case study on the PlantCare ubiquitous computing application. The benefits of connector abstractions include separating communication code from application logic, documenting and checking connector interfaces, and reusing connector abstractions more effectively compared with alternative techniques

In future work, we intend to implement more connectors and evaluate their expressiveness on a wider variety of systems. We also hope to develop a library-based framework for composing connectors together so that complex connectors can be easily created from simple building blocks. Another important area of future work is more effective support for adaptor-style connections, extending recently developed adaptation techniques such as on-demand remodularization [MO02]. Finally, we would like to provide specification and checking of connector properties that go beyond typechecking techniques. We believe that enhanced language and system support for connectors is crucial to the effective development and evolution of many classes of software systems.

Acknowledgements

We would like to thank members of the Cecil and 590N seminars at the University of Washington, as well as Anthony LaMarca, Stefan Sigurdsson, Matt Lease, and other members of the PlantCare group at Intel Research Seattle for their help, comments, and suggestions. This work was supported in part by NSF grants CCR-9970986 and CCR-0073379, **NEW GRANT**, and gifts from Sun Microsystems and IBM.

References

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

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

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

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

[BA01] Lodewijk Bergmans and Mehmet Aksit, Composing Crosscutting Concerns Using Composition Filters, Communications of the ACM 44(10):51-57, October 2001.

[BRP02] Chrislain Razafimahefa, Ciaran Bryce, and Michel Pawlak. Lana: An Approach to Programming Autonomous Systems. Proc. European Conference on Object-Oriented Programming, Malaga, Spain, June 2002.

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

[DMT99] Eric M. Dashofy, Nenad Medvidovic, and Richard N. Taylor. Using Off-the-Shelf Middleware to Implement Connectors in Distributed Software Architectures. Proc. International Conference on Software Engineering, Los Angeles, California, May 1999.

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

[HK02] Jan Hannemann and Gregor Kiczales. Design Pattern Implementation in Java and AspectJ. Proc. Object-Oriented Programming Systems, Languages, and Applications, Seattle, Washington, November 2002.

[Jav97] Javasoft Java RMI Team. Java Remote Method Invocation Specification, Sun Microsystems, 1997.

[JLH88] Eric Jul, Hank Levy, Norman Hutchinson, and Andrew Black. Fine-Grained Mobility in the Emerald System. ACM Trans. Computer Systems 6(1):109-133, February 1988.

[KRB91] Gregor Kiczales, James des Rivières, and Daniel G. Bobrow. The Art of the Meta-Object Protocol. MIT Press, Cambridge, MA, 1991.

[LBK+02] A. LaMarca, W. Brunette, D. Koizumi, M. Lease, S. B. Sigurdsson, K. Sikorski, D. Fox, and G. Borriello. PlantCare: An Investigation in Practical Ubiquitous Systems. In UbiComp ’02.

[LC02] Keunwoo Lee and Craig Chambers. Hydro: A Language for Loosely Coupled Ubiquitous Systems. Submitted for publication, 2002.

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

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

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

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

[MMP00] Nikunj R. Mehta, Nenad Medvidovic, and Sandeep Phadke. Towards a Taxonomy of Software Connectors. Proc. International Conference on Software Engineering, Limerick, Ireland, June 2000.

[MO02] Mira Mezini and Klaus Ostermann. Integrating Independent Components with On-Demand Remodularization. Proc. Object-Oriented Programming Systems, Languages, and Applications, Seattle, Washington, November 2002.

[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.

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

[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.

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

[RS02] Ran Rinat and Scott Smith. Modular Internet Programming with Cells. Proc. European Conference on Object-Oriented Programming, Malaga, Spain, June 2002.

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

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

[SDZ96] Mary Shaw, Rob DeLine, and Gregory Zelesnik. Abstractions and Implementations for Architectural Connections. Proc. International Conference on Configurable Distributed Systems, Annapolis, Maryland, May 1996.

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