Exogenous Connectors for Software Components

Kung-Kiu Lau[*], Perla Velasco Elizondo, and Zheng Wang

School of Computer Science, The University of Manchester
Manchester M13 9PL, United Kingdom
{kung-kiu,pvelasco,zw}@cs.man.ac.uk

Abstract. In existing component models, control originates in components, and connectors are channels for passing on the control to other components. This provides a mechanism for message passing, which allows components to invoke one another's operations by method calls (or remote procedure calls) either directly or indirectly via a channel such as a bus. Thus components in these models mix computation with control, since in performing their computation they also initiate method calls and manage their returns, via connectors. Consequently, in terms of control, components are not loosely coupled. In this paper, we propose *exogenous* connectors, and demonstrate their use in a small example. In contrast to connectors in existing component models, exogenous connectors initiate calls and manage their returns, and are used to encapsulate control in a component model we are working on. In the example, we demonstrate the feasibility of exogenous connectors, and compare them with connectors in closely related architecture description languages.

1 Introduction

Components and connectors are the basis of many software component models. Architecture Description Languages (ADLs) [22] have always defined software systems in terms of components (boxes) and connectors (lines) that link components and thus define relationships between them. More recently, UML 2.0 [19] also uses connectors to compose components. Even component models that do not use connectors explicitly often have composition operators that can be interpreted as connectors at different levels of abstraction. Lower-level connectors act as wiring mechanisms, while higher-level connectors can correspond to sophisticated protocols or control structures. For example, direct method calls between components may be regarded as code-level connectors, and glue code or scripts [21] that combine components can be regarded as interface-level connectors. A large and detailed taxonomy of software connectors can be found in [16].

In existing component models, connectors are meant to encapsulate *interaction* or *communication* while components are meant to encapsulate *computation*. In these models, control originates in components, and connectors are channels for coordinating the control flow (as well as data flow) between components. This provides a mechanism for message passing, which allows components to invoke one another's operations by method calls (or remote procedure calls) either directly or indirectly via a channel such as a bus. Thus components in these models mix computation with control, since in

[*] Partially supported by CoLogNet, a European Network of Excellence funded by IST FET.

performing their computation they also initiate method calls and manage their returns, via connectors. Consequently, in terms of control, components are not loosely coupled. In particular, although they encapsulate communication, connectors do not encapsulate control: they only pass it on.

In this paper, we introduce *exogenous* connectors. These connectors are different in that they encapsulate control flow between components *totally*, i.e. they originate and coordinate *all* control. This means that components do not invoke methods or procedures in other components via these connectors; rather, this is done by the connectors.

Our main motivation for using exogenous connectors is to encapsulate control in a component model we are working on, in order to minimize coupling between components. In our model, components encapsulate data and functions, and are therefore loosely coupled in terms of these. Using exogenous connectors to encapsulate control completes the encapsulation that we wish to achieve, and thereby maximizes loose coupling, i.e. in terms of data, functions *and* control. A corollary of such complete encapsulation in our component model is that reasoning about components and their composition should become more tractable and hence practicable. This offers hope that our component model could have the capability for predictable assembly [23].

2 Exogenous Connectors

In this section, we introduce exogenous connectors as defined in our component model. By way of contrast, we first consider connectors in current component models, and briefly assess them for encapsulation and loose coupling.

Connectors in current component models fall into two main categories: (i) connectors that represent composition by *direct* message passing; and (ii) connectors that represent composition by *indirect* message passing. In these models, components are usually software units (typically classes or objects) with their own methods or functions which can be invoked by other components either directly by method calls or indirectly via code that links the components together.

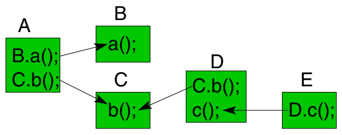


Fig. 1. Connecting components by direct message passing.

2.1 Connecting Components by Direct Message Passing

Connecting components by direct message passing is illustrated in Fig. 1. For convenience we borrow the dot notation from object-oriented languages for components and their methods. For example, in Fig. 1, if component A calls the method a of component

B, then it does so by passing a message directly to B. In a message passing scheme, there are two distinct roles: the *sender* and the *receiver* of a message. The identity of the receiver is either statically known to the sender or it is dynamically evaluated at execution time. Sometimes there can be more than one receiver, as for instance, the message may be multi-cast to several receivers or broadcast to all receivers in the system. From the sender's point of view, the identity of the intended receiver is known a priori, but the receiver does not have to know the sender at all. Thus the send operation is generally targeted by the sender at a specific set of receivers. Remote Procedure Calls (RPC), method and event delegation are well-known examples of message passing schemes. Software component models that adopt message passing schemes as composition operators are Enterprise JavaBeans [13], CORBA Component Model [20], COM [8], UML [19] and KobrA [6].

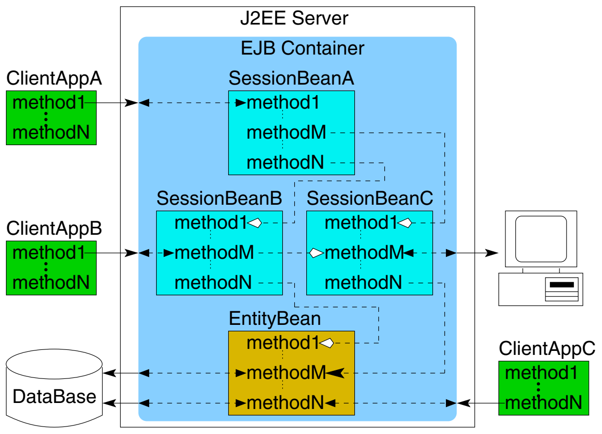


Fig. 2. Connecting beans by direct method calls in Enterprise JavaBeans.

For example, in Enterprise JavaBeans (EJB), the beans are Java classes in an EJB container that are connected by direct method calls, as illustrated by the example in Fig. 2. Even client applications are connected to the beans by method calls via the EJB container. In general, when components are connected by direct message passing, communication expressing control is mixed with computation, and sender components and their receivers are tightly coupled with each other. The worst problem in this regard is reentrant calls, when a method y called by a method x calls x itself. Also, there is no explicit code for connectors, since messages are 'hard-wired' into the components, and so these connectors are not separate entities and therefore cannot be reused. In particular, they cannot be pre-defined and deposited in a repository.

2.2 Connecting Components by Indirect Message Passing

Connecting components by *indirect* message passing is illustrated in Fig. 3. Here, connectors are separate entities that are defined explicitly. Typically they are glue code or scripts that pass messages between components indirectly. To connect a component to another component we use a connector that when notified by the former invokes a method in the latter. For example, in Fig. 3, component A is connected to component B by connector $Con1$, so whenever A sends a message to notify $Con1$, the latter passes a message to component B to invoke method a in B. In JavaBeans [9], for example, beans are connected precisely in this way, using adaptor classes as connectors. This kind of connector is at a slightly higher level of abstraction (and indirection) than direct method calls, but is nevertheless still rather low-level, since it essentially glues or wires components together.

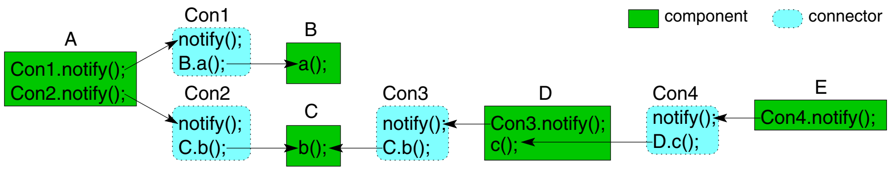


Fig. 3. Connecting components by indirect message passing.

More abstractly, when connected by indirect message passing, components can be viewed as computational units with *in* and *out* ports, and connectors connect matching ports to pass control as well as data between components. For example, in Fig. 3, component A has two out ports, B has one in port, and the connector $Con1$ connects one of A's out ports to B's in port. In ADLs, components are connected together by connectors via ports in precisely this way. For example, in Acme [12], Fig. 3 would be drawn as Fig. 4 (a), where ports are represented by triangles and connectors by two lines joined by a black dot.

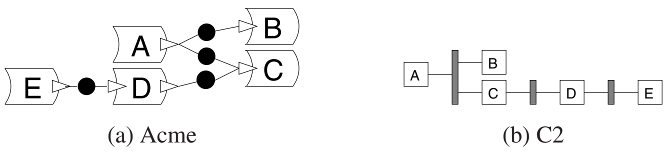


Fig. 4. Connecting components in ADLs.

In ADLs, a component represents a primary computational element and data store of a system. The interface of a component is defined as a set of ports through which its functionalities are exposed. Components are connected by connectors that link their ports. The connectors mediate the communication and coordination activities among components. Typical connectors in ADLs are pipes used by components to pass information from one to another. Such information may be simple data values or messages for invoking methods.

In messaged-based ADLs, it is possible to have a still higher level of abstraction of indirect message passing. This is exemplified by C2 [25], where components are linked to a bus, and messages can be passed between components by broadcasting[1] them on the bus. For example, in C2, Fig. 3 would be drawn as Fig. 4 (b). The links are not directed, so it is not possible to say explicitly which component calls which. Instead, messages are placed on the bus and components linked to the bus must determine which messages are intended for them. Besides JavaBeans and ADLs, other software component models that adopt indirect message passing schemes are Koala [26], PIN [15] and PECOS [18].

In general, when components are connected by indirect message passing, communication (connectors) is separated from computation (components). However, this does not necessarily mean that control is separated from computation, since the information passed from and to a component may contain method calls. As a result, components are tightly coupled by connectors.

Although they are separate entities, connectors are usually not intended to be defined and deposited in a repository. Rather, they are pieces of code generated for specific sets of components. Therefore, these connectors are not reusable. For example in JavaBeans, only beans can be stored in a repository, but not connectors. The latter have to be generated (automatically by the builder tool) as adaptor classes for each specific pair of bean instances. In ADLs, both components and connectors are meant to represent design and not stored in a repository.

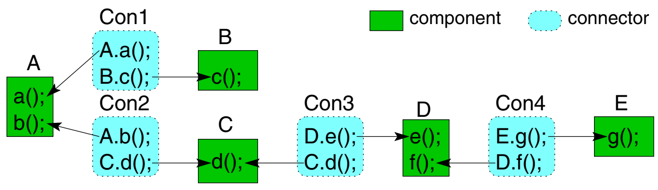


Fig. 5. Connecting components by exogenous connectors.

2.3 Connecting Components by Exogenous Connectors

Using connection by message passing, both direct and indirect, connected components invoke each other's methods, so their connection effects control, and possibly data, flow as well as computation. Connected components are thus tightly coupled, albeit to a varying degree depending on the level of indirection in the message passing; and control and computation are mixed up. In order to minimize coupling, and to maximize separation of control from computation, we propose exogenous connectors which encapsulate control and data flow between connected components. The idea is that in connected components, the connectors, rather than the components themselves, initiate method calls in the components, and handle any accompanying data flow, so that any

[1] Other modes of communication are possible in C2, but broadcasting is the most indirect.

control flow between the components is encapsulated by the connectors. This is illustrated in Fig. 5. In such a scheme, connected components react to their connector only, and not directly with each other. Components encapsulate computation, while the connectors encapsulate control. For example, in Fig. 5, the connector $Con1$ calls method a in component A and method c in B, but A and B do not directly interact with each other. Method calls may be accompanied by data flow between A and $Con1$ or between B and $Con1$, but again not between A and B directly.

Connection Scheme	Component Models	Control mixed with computation	Calls initiated from outside components
Direct message passing	EJB, CCM, COM, UML, KobrA	✓	×
Indirect message passing	JavaBeans, ADLs, PECOS, Koala, PIN	✓	×
Exogenous connection	Our proposed model	×	✓

Fig. 6. Comparison of connection schemes.

Fig. 6 is a comparison of exogenous connectors with connectors used in existing component models. It shows clearly the contrast between exogenous connectors and non-exogenous ones: the former do not mix control and computation, whereas the latter do; using the former, method calls are initiated from outside components, whereas using the latter, they are initiated by components themselves. In the component model we are working on, we plan to use exogenous connectors for composition.

Although no existing component model uses it, exogenous connection has been defined as exogenous coordination in coordination languages for concurrent computation [3]. Also, in object-oriented programming, the courier pattern [11] uses the idea of exogenous connection whereby a courier object links a producer-consumer pair of objects by calling the *produce* method in the producer object and then calling the *consume* method in the consumer object with the result of the *produce* method.

3 Creating and Using Exogenous Connectors

In a component model we are working on, we want to use exogenous connectors as composition operators. In this section, we identify the types of exogenous connectors that we will need, and show how we create and use them in a preliminary implementation of our component model. First we outline the relevant aspects of our model.

In our component model, a component is a unit of software with (i) an *interface* that specifies the services it provides and the services it requires, and the dependencies between the two sets of services; and (ii) *code* that implements the provided services. In essence it is similar to Szyperski's definition [24]. However, components do not request services in other components. Rather, they perform their provided services only when invoked externally by connectors. Thus components encapsulate computation.

Connectors are composition operators that compose components into systems. They are in essence similar to connectors in ADLs, except of course that they are exogenous.

They initiate and coordinate method calls in components and handle their results. Thus they determine control flow and data flow, i.e. they encapsulate communication.

An important feature of our component model is the definition of the life cycle of components as consisting of two stages: (i) *repository*, or *design*, phase, and (ii) *deployment*, or *execution*, phase. In the repository phase, components as well as connectors have to be constructed, catalogued and stored in a repository in such a way that they can be retrieved later, as and when needed. Components in the repository are templates that are stateless. In the deployment phase, components are retrieved from the repository, and instantiated with initial data. Therefore components have *states* and are ready for execution. Similarly, connectors are retrieved and instantiated, and ready to execute.

In a preliminary implementation of our component model, for simplicity and for ease of implementation, components and connectors are defined as Java classes in the repository phase, so that instances can be created and deployed in the deployment phase. We do not yet address issues related to interfaces or repository management.

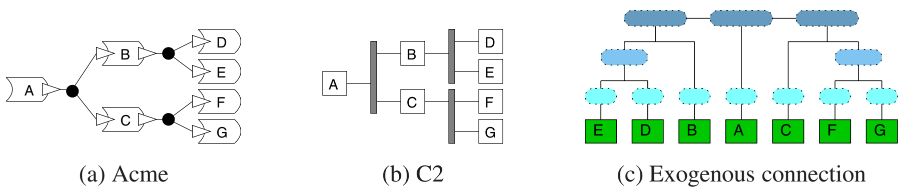


Fig. 7. Example architecture using exogenous connection, and equivalent ADL architectures.

3.1 Types of Exogenous Connectors

Since, in our component model, objects implementing components are not allowed to call methods in other components, we need an exogenous *method invocation connector*. This is a *unary* operator that takes a component, invokes one of its methods, and receives the result of the invocation. To structure the control and data flow in a set of components or a system, we need other connectors for sequencing exogenous method calls to different components. So we need n-*ary* connectors for connecting invocation connectors, and n-*ary* connectors for connecting these connectors, and so on. In other words, we need a hierarchy of connectors of different arities and types. This is illustrated by the example architecture in Fig. 7 (c), which represents a system that can be described in Acme and C2 by the respective architectures in Fig. 7 (a) and (b). In Fig. 7 (c), the lowest level of connectors are unary invocation connectors that connect to single components, the second-level connectors are binary and connect pairs of invocation connectors, and the third-level connectors are of variable arities and types.

In general, connectors at any level other than the first can be of variable arities; connectors at any level higher than two can be of variable arities *and* types; and we can define any number of levels of connectors. Connectors at level n for any $n > 1$ can be defined in terms of connectors at levels 1 to $(n-1)$, according to the following type hierarchy (omitting methods and their parameters):

$$\begin{array}{lll}
\textit{Basic types} & \text{Component, Result;} & \\
\textit{Connector types} & L1 \equiv \text{Invocation} & \equiv \text{Component} \longrightarrow \text{Result;} \\
 & L2 & \equiv L1 \times \ldots \times L1 \longrightarrow \text{Result;} \\
 & L3 & \equiv L \times \ldots \times L \longrightarrow \text{Result} \\
 & \ldots & \quad \text{where } L \text{ is either } L1 \text{ or } L2;
\end{array}$$

Thus level-one and level-two connectors are not polymorphic, but connectors at higher levels are. More formally, for an arbitrary number n of levels, the connector type hierarchy can be defined in terms of dependent types and polymorphism as follows:

$$\begin{array}{rl}
 & L1 \equiv \text{Component} \longrightarrow \text{Result;} \\
 & L2 \equiv L1 \times \ldots \times L1 \longrightarrow \text{Result;} \\
\text{For } 2 < i \leq n, & Li \equiv L(j_1) \times \ldots \times L(j_m) \longrightarrow \text{Result, for some } m \\
 & \quad \text{where } j_k \in \{1, ..., (i-1)\} \text{ for } 1 \leq k \leq m, \\
 & \quad \text{and } L(i) = \begin{cases} L1 \ , \ i = 1 \\ \vdots \\ Ln \ , \ i = n. \end{cases}
\end{array}$$

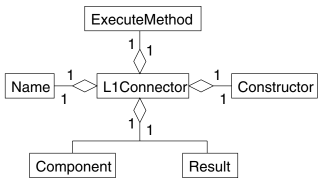
3.2 Implementing Exogenous Connectors

Having defined the types of the hierarchy of connectors for our component model, we need to find a way to implement connectors of these types in a generic way, such that: (i) in the repository phase, connector *templates* can be defined and stored in a repository (along with components); (ii) in the deployment phase, connector *instances* can be created (and deployed with component instances).

```
package connectors;
import java.lang.reflect.*;
public class Connector {
  public void execute (Method m, Object [] params) {}
  public void execute (Method[] ms, Object [] params) {}
}
```

Fig. 8. The *Connector* superclass.

We can do so by implementing the connectors as a hierarchy of Java classes, with a superclass *Connector* (Fig. 8). The *Connector* class has two *execute* methods for executing either a single given method (with its parameters) or a given set of methods (with their parameters). The Method class in the *execute* methods of *Connector* is provided by Java reflection. It provides the *invoke* method for calling method instances (see later). Using the *Connector* class, we can define a generic connector at any level of the hierarchy. Such a connector inherits from *Connector*, and implements the appropriate *execute* method(s). Any desired instances of this connector can then be created.



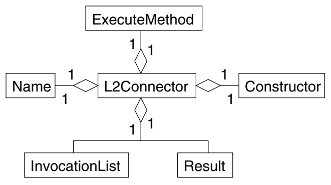
(a)

```
package connectors;
import java.lang.reflect.*;
public class Invocation extends Connector {
  private Object component;
  public Object result;
  public Invocation (Object c) { ... }
  public void execute (Method m, Object [] params) {
    this.result = m.invoke(component, params);
} }
```

(b)

Fig. 9. Level-one (invocation) connectors: design and implementation.

A level-one connector is a *unary* invocation connector. The general design of level-one connectors is shown in Fig. 9 (a). Each connector *L1Connector* (with a unique *Name* and a *Constructor*) connects a single component. It executes a method *ExecuteMethod* that can invoke any method in that component, and yields a single *Result*. This design can be implemented in Java using reflection, as the invocation connector in Fig. 9 (b). The *execute* method of the invocation connector uses the *invoke* method provided by Java reflection to call the chosen method.

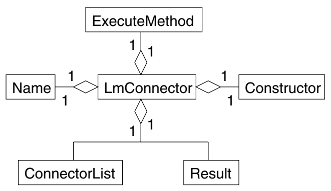


(a)

```
package connectors;
import java.lang.reflect.*;
public class Pipe extends Connector {
  private Invocation[] invs;
  public Object result;
  public  Pipe (Invocation[] is) { ... }
  public void execute (Method[] ms, Object [] params) {
    ... invs[0].execute(ms[0], params); tmp = invs[0].result;
    ... invs[invs.length].execute(ms[invs.length], tmp);
    this.result = invs[invs.length].result;            }    }
```

(b)

Fig. 10. Level-two connectors: design and implementation.



(a)

```
package connectors;
import java.lang.reflect.*;
public class Pipem extends Connector {
  private Connector[] cons;
  public Object result;
  public  Pipem  (Connector [] cs) { ... }
  public void execute (Method[] ms, Object [] params) {
    ... cons[0].execute(ms[0], params); tmp = cons[0].result;
    ... cons[cons.length].execute(ms[cons.length], tmp);
    this.result = cons[cons.length].result           } }
```

(b)

Fig. 11. Level-m connectors: design and implementation.

Level-two connectors are n-*ary* connectors that connect invocation connectors (Fig. 10 (a)). There are different kinds of connectors, with different implementations of their *ExecuteMethods*. For example, an n-*ary pipe* connector, used to pass values successively from the execution of a method of one component to the input of a method of the next component, can be implemented in the manner outlined in Fig. 10 (b). Another example of a level-two connector is a n-*ary selector* connector that selects one connector to execute. The *execute* method of a selector would first define how to choose the connector, before calling the *execute* method of the latter. Thus, for level-two connectors, the implementation of the *execute* method is connector-specific. However, the implementation technique is the same for all these connectors, since they are defined in terms of level-one connectors, which are implemented using Java reflection.

For an arbitrary level $m > 2$, connectors are n-*ary* and connect connectors of levels lower than m. Unlike level-one and level-two connectors, these connectors are polymorphic, so choices have to be made depending on the application in question. In general the design for level m connectors is shown in Fig. 11 (a), where a connector connects a list *ConnectorList* of connectors. Fig. 11 (b) shows the outline of the implementation of a level-m pipe.

4 An Example: The Bank System

Having defined and implemented the hierarchy of exogenous connectors that we need for our component model, we now demonstrate their use in a simple application, and use it to compare our approach with closely related work, viz. the Acme and C2 ADLs.

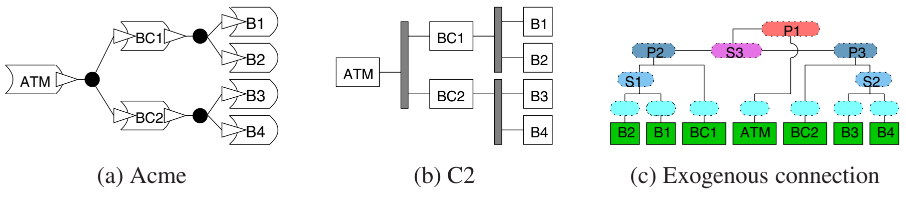


Fig. 12. Architectures of the bank example.

The example we have chosen is a simple bank system, whose architecture is described in ACME and C2 in Fig. 12 (a) and (b) respectively. The system has just one ATM that serves two bank consortia (BC1 and BC2), each with two bank branches (B1 and B2, B3 and B4 respectively). The ATM passes customer requests together with customer details to the customer's bank consortium, which in turn passes them on to the customer's bank branch. The bank branches provide the usual services of withdrawal, deposit, balance check, etc.

4.1 Implementation Using Exogenous Connectors

In our component model, using exogenous connectors, we implemented the architecture in Fig. 12 (c) for the bank system. The first step is to implement the components, and the second step is to construct a structure of connectors, i.e. a control structure, to sit on top of the components. The connector structure is constructed level by level. At level one, an invocation connector is connected to every component. This enables all the methods of a component to be invoked. Then at level two, invocation connectors are connected by level-two connectors which effect appropriate behavior among the components connected to the invocation connectors. At level three, level-two connectors are connected by level-three connectors, and so on. Execution of the system starts at the connector at the highest level; this connector is the one to initiate control flow.

In our implementation of the bank example (Fig. 12 (c)), components are Java objects with public methods (that can be invoked by the invocation connectors). These objects do not call methods in other components.

At level two, there is a selector connector $S1$ that is used to select the customer's bank branch from banks $B1$ and $B2$, prior to invoking that branch's methods requested by the customer. Similarly, there is a level-two selector connector $S2$ for choosing between $B3$ and $B4$, prior to invoking their methods requested by the customer. To pass values from one bank consortium to one of its banks we need a pipe connector; at level three, we have two pipe connectors $P2$ and $P3$, for $BC1$ and $BC2$ respectively. At level four, $S3$ is a selector connector that selects the customer's bank consortium from consortia $BC1$ and $BC2$. Finally, at level five, the pipe connector $P1$ initiates the banking system's operational cycle by passing customer requests and card information to the *ATM*, invoking the *ATM*'s methods, and then passing resulting value to connector $S3$.

```
package system;
import java.lang.reflect.*;
import java.io.*;
import connectors.*;
public class BankSystem  {
 public static void main (String[] args) {
  // create instances of components
  ATM atm = new ATM("1"); ... Bank bank4 = new Bank("4");
  // create level-one connectors
  Invocation invATM = new Invocation((Object) atm)); ... Invocation invB4 = new Invocation((Object) bank4));
  // create level-two connectors
 Invocation[] invsBank12 = new Invocation[2]; invsBank12[0] = invB1; invsBank12[1] = invB2;
 Selector s1 = new Selector(invsBank12);
 // create level-three connectors
 Connector[] consBC1 = new Connector[2];  consBC1[0] = invBC1;  consBC1[1] = s1;
 Pipem p2 = new Pipem(consBC1); ... Pipem p3 = new Pipem(consBC2);
 // create level-four connectors
 Connector[] consAB = new Connector[2]; consAB[0] = p2; consBC[1] = p3;
 Selectorm s3 = new Selectorm(consAB);
 // create level-five connectors
 Connector[] consm = new Connector[2]; consm[0] = invATM; consm[1] = s3;
 Pipem p1 = new Pipem(consm);
 // Display menu and initiate operations
 switch(Integer.parseInt(args[0]))) {
  case 1:
   System.out.println("Your balance is:");
   p1.execute(ms, params);
   break; ...                   } } }
```

Fig. 13. Outline of the code for the bank system.

Fig. 13 shows the outline of the code for the system. It reflects the hierarchical manner in which the bank system is built, by constructing the connectors level by level. We can illustrate the bank system's behavior by considering the example of a customer of $B3$ wishing to withdraw money from his account. First the customer inserts his card into the teller machine and keys in his PIN code. The $P1$ pipe connector initiates the system's operational cycle by passing the customer information and withdrawal request to the invocation connector of the *ATM* component to invoke the appropriate methods of the *ATM* for validating customer details and determining the customer's bank consortium. If the validation succeeds, $P1$ passes the bank consortium identity, $BC2$ (and the withdrawal request), to the selector $S3$. $S3$ selects the pipe $P3$, which is connected to $BC2$. $P3$ uses the invocation connector for $BC2$ to invoke a method for identifying the customer's bank branch, and passes the result, $B3$, to the selector $S2$ connector. $S2$ selects the invocation connector of $B3$ to invoke the *withdraw* method implemented by $B3$. The result is then the output of this operational cycle. It can be output by either the selector $S2$, or even the component $B3$.

4.2 Comparison with Acme and C2 Implementations

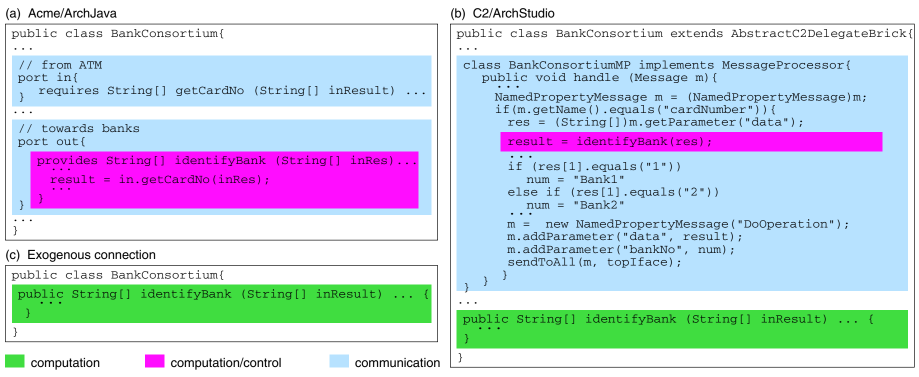
The implementation of the bank example demonstrates the feasibility of using exogenous connectors (and our component model) to build systems. In order to evaluate our approach against related work, we implemented the same example using Acme and C2, and compared these implementations with ours. We chose Acme because it is the most generic archetypal ADL. We chose C2 because, as a message-based ADL, it uses a higher level of abstraction of indirect message passing than non-message-based ADLs such as Acme, as explained in Section 2.2; thus connection in C2 is more indirect than Acme. These two provide a graded comparison with exogenous connection.

Another reason for choosing Acme and C2 is a practical one, namely that there are tools for generating implementations from architecture descriptions in these ADLs. The tools we chose are ArchJava [4,2] and ArchStudio 3 [5] for Acme and C2 respectively. Both these tools are based on Java, and generate Java code, so they allow easy and direct comparison with our example.

The main point of comparison is the separation of control and communication from computation. In our component model, components are supposed to encapsulate computation while exogenous connectors are supposed to encapsulate communication in general and control in particular. This distinguishes it from existing component models, in particular ADLs like Acme and C2, as shown in Fig. 6. Our experiment with the bank example bears this out very well.

At design time, as shown in Fig. 12, Acme and C2 architectures look 'as exogenous as' our approach. However, when the architectures are implemented, the resulting Acme/ArchJava and C2/ArchStudio systems show clearly the mixing of control and computation, and of computation and communication, whereas our system maintains their separation. Fig. 14 shows the structure of a BankConsortium (BC) component in Acme/ArchJava, C2/ArchStudio and our approach, distinguishing between code for computation, code that mixes computation and control, and code for communication.

In the case of Acme/ArchJava, Fig. 14 (a), components communicate through *ports* that must be defined in the component's class. Required operations are specified in *in*


(a) Acme/ArchJava

```
public class BankConsortium{
...
 // from ATM
 port in{
 }  requires String[] getCardNo (String[] inResult) ...
...
 // towards banks
 port out{
    provides String[] identifyBank (String[] inRes)...
      ...
      result = in.getCardNo(inRes);
      ...
    }
 }
...
}
```

(b) C2/ArchStudio

```
public class BankConsortium extends AbstractC2DelegateBrick{
...
 class BankConsortiumMP implements MessageProcessor{
    public void handle (Message m){
      ...
      NamedPropertyMessage m = (NamedPropertyMessage)m;
      if(m.getName().equals("cardNumber")){
        res = (String[])m.getParameter("data");
        result = identifyBank(res);
        ...
        if (res[1].equals("1"))
           num = "Bank1"
        else if (res[1].equals("2"))
           num = "Bank2"
        ...
        m =  new NamedPropertyMessage("DoOperation");
        m.addParameter("data", result);
        m.addParameter("bankNo", num);
        sendToAll(m, topIface);
      }
    }
 }
...
 public String[] identifyBank (String[] inResult) ... {
   ...
 }
}
```

(c) Exogenous connection

```
public class BankConsortium{
 public String[] identifyBank (String[] inResult) ... {
   ...
  }
}
```

Fig. 14. Computation, control and communication in the BankConsortium component.

ports and provided operations are defined in *out* ports. Thus the provided operations are defined in the communication part of the component, i.e. mixing communication and computation. Moreover, by definition, the provided operations are defined in terms of the required operations, i.e. the former will call the latter, thus mixing computation with control. In the bank example, in the computation part (in the communication part) of the BankConsortium component, the provided operation *identifyBank* computes *result* by calling the required *getCardNo* operation named in an *in* port, and thus directs control flow to the (connector to the) ATM component, that provides this operation.

In the case of C2/ArchStudio (Fig. 14 (b)) to receive and send messages, a component must implement its own message processor. Message processors are objects that implement a *handle* method to deal with messages. In the bank example, the BankConsortium component uses the *BankConsortiumMP* inner class as its message processor. A message processor must identify messages intended for the component, and determine and execute appropriate actions upon receipt of such messages, which may include sending a message with the results of the actions. Here, communication is mixed up with computation that contains control. In the bank example, the message processor *BankConsortiumMP* computes *result* by calling the *identifyBank* method in the midst of its communication code, and thus directing control flow to this method.

By contrast, in our approach, Fig. 14 (c), the separation of computation from control and communication is maintained. A component has only code for computation. In the bank example, the BankConsortium component has only code for the *identifyBank* method. Communication, and control in particular, is embodied in connectors.

Another point of comparison is the supported patterns of communication, i.e. the set of connectors provided. Compared to Acme and C2, we have a potentially larger set of connectors, since our hierarchy of connector types is polymorphic and can be used to generate any number of kinds of connectors at any level. In contrast, Acme has just one level of connectors, viz. pipes, for indirect message passing such as remote procedure calls via components' ports; C2 uses a bus for connecting components at every level.

5 Evaluation and Discussion

The example in the previous section provides some useful initial feedback on exogenous connectors. In particular, the comparison of the implementation with those in Acme/ArchJava and C2/ArchStudio shows that exogenous connectors can offer some advantages. While it would be unwise to draw general conclusions from one small example, it is possible to make some general observations and discuss their import. In this section we evaluate exogenous connectors and discuss their potential usefulness, advantages and disadvantages, etc. with respect to CBSE.

The bank example demonstrates the feasibility of using exogenous connectors to encapsulate communication, in particular control flow. Separating computation from control means that control flow does not originate from components, but from connectors. So in a system, the components are decoupled from the structure of connectors which provides the control structure. For system maintenance and evolution, this decoupling should make it simpler to manage changes in the components and changes in the connectors separately. Another advantage should be separation of concerns in reasoning about system behaviors. The connectors encapsulate the computational paths, while the components encapsulate computation. In predictable assembly, this separation of concerns should make it more tractable and easier to reason about system behavior by reasoning about control and computation separately.

An important implication of this is that it should be possible to verify component properties, e.g. functional correctness, statically and store proven components in a repository. To this end, we plan to develop components with contracts [17], statically prove their contract compliance, and then place them in the repository. At deployment time, the developer can assume the components' contract compliance, and concentrate on developing the connector structure, and reasoning about its computational paths. Again, separating control from computation should make it more tractable and hence practicable to reason about components and their composition. Such an approach would make a fundamental contribution to CBSE, since it enables bottom-up assembly of components, starting with a component repository. The only existing component models that support this are Koala [26] and KobrA [6], which have repositories for product-lines. Our approach is different in that the repository components form a flat tier at the bottom, and the control structure is built to sit on top, as clearly illustrated by Fig. 12 (c). By contrast, top-down design approaches, e.g. ADLs, do not use components from repositories, and do not allow composites to be constructed and stored in repositories for reuse in different applications. In fact, even our exogenous connectors can be stored in a repository, as we saw in the bank example.

Another practical advantage of exogenous connectors is their hierarchical nature. This provides a hierarchical way for developing systems, resulting in well-structured code for the final system, which is easy to understand, and therefore maintain. Adding, changing and replacing connectors is also made easier. It remains to be seen, however, whether these advantages pertain, or whether our approach is practical, when the application is very large, requiring a very sophisticated and involved control structure. It would be interesting to investigate the design and implementation of more complicated exogenous connectors than pipes and selectors, and whether their existence would make our approach practical for such applications.

The obvious potential disadvantage of exogenous connectors is a potential preponderance of connectors and connector levels, and hence inefficiency in communication. In the bank example, the number of levels of connectors is only five. Considering the architecture is a three-level architecture in Acme and C2, this is not bad. However, we have not studied how to predict the number of levels, nor the total number, of connectors. Also, to measure run-time performance sensibly, we need much larger examples.

6 Conclusion

In this paper we have presented exogenous connectors, their definition and implementation, and an example illustrating their use in building a simple system. The key distinguishing characteristic of these connectors is that they encapsulate control totally, i.e. control originates from them. This is in complete contrast to most software connectors, which encapsulate communication but not control. In ADLs, for example, control originates in components, and connectors pass it on to other components, and so on.

Our work on exogenous connectors is at a preliminary stage, however. We have considered only control flow, but not data flow, beyond passing results of method calls back to connectors. Data flow needs careful consideration, especially if components have private databases, or if some but not all components in a system share databases. In such systems, the precise ways in which global data and local data interact must be clearly defined and rigorously enforced.

We have not considered concurrency in the connectors either. In fact we have only used sequential composition. As a result, we cannot address the general issues of architectural reasoning, whereby typically connectors provide parallel or asynchronous communication between components. For example, one such issue is that of "correct architectures" in the sense of architectures with guaranteed temporal properties such as deadlock-freedom [14]; another issue concerns consistency between architecture and code, e.g. [1]. However, by using only sequential composition, we manage to avoid the problem of synchronization, which is present in even the simplest forms of message passing, such as direct method calls, when there are competing calls to the same callee.

The connectors we have presented in this paper belong to the deployment phase of our component model. We are in the process of investigating connectors for the other phase, the repository phase. In this phase, we want to store components as well as connectors. Our aim is to also allow the construction of composite components, and to store them in the repository, so as to make the task of composition in the deployment phase simpler. Top-down approaches such as software architectures do not use components from, or store them, in repositories. In ADLs, although they can be constructed and used at design time, composite components are not stored in repositories. By contrast, we want to do so. We believe that in the repository phase, it would be useful to have connectors that can be used to compose components into subsystems or even complete systems that can be stored in the repository. Such connectors would have to yield components from those they connect. The invocation connectors, pipes and selectors in this paper return results rather than components, and are therefore deployment phase connectors, and cannot be used to build composites in the repository phase.

In this paper, we have implemented components as Java classes. This is of course only for convenience. In our component model, components are not necessarily just classes. In particular, they have proper interfaces that may not be implementable using classes in object-oriented programming languages. For example, our interfaces contain contracts which specify the behavior of the components, apart from the signature of its operations. These contracts cannot be defined properly in Java using the *assert* statement, which is the only provision in Java for Design by Contract.

Apart from the issue of interfaces, Java is a good implementation language, or at least as good as any object-oriented language. Inheritance allows us to define our polymorphic connector hierarchy straightforwardly. However, the problems with inheritance are well-known, not least of which is any form of static analysis, which is important for reasoning purposes in the repository phase. In particular, inheritance works against our wish to statically verify component and connector contracts in the repository phase.

For predictable assembly, we believe the use of contracts in the repository phase is crucial, as is their static verification, and we have started investigating using ESC Java [10] for both components and connectors. We are also investigating other languages, in particular SPARK [7], which offer tool support for contracts and their static verification. Finally, in the longer term, it would be interesting to examine the taxonomy in [16] to see how widely the idea of exogenous connectors can be applied to that taxonomy.

Acknowledgements

We wish to thank David Garlan for helpful points of information about code generation from architecture descriptions.

References

1. J. Aldrich, C. Chambers, and D. Notkin. Architectural reasoning in ArchJava. In *Proc. 16th European Conference on Object-Oriented Programming*, pages 334–367. Springer-Verlag, 2002.
2. J. Aldrich, C. Chambers, and D. Notkin. ArchJava: Connecting software architecture to implimentation. In *Proc. 24th International Conference on Software Engineering*, pages 187–197. ACM Press, 2002.
3. F. Arbab. The IWIM model for coordination of concurrent activities. In P. Ciancarini and C. Hankin, editors, *Lecture Notes in Computer Science 1061*, pages 34–56. Springer-Verlag, 1996.
4. ArchJava web page. http://archjava.fluid.cs.cmu.edu/index.html.
5. ArchStudio 3 web page. http://www.isr.uci.edu/projects/archstudio/index.html.
6. C. Atkinson, J. Bayer, C. Bunse, E. Kamsties, O. Laitenberger, R. Laqua, D. Muthig, B. Paech, J. Wüst, and J. Zettel. *Component-based Product Line Engineering with UML*. Addison-Wesley, 2001.
7. J. Barnes. *High Integrity Software: The SPARK Approach to Safety and Security*. Addison-Wesley, 2003.
8. D. Box. *Essential COM*. Addison Wesley, 1998.
9. R. Englander. *Developing Java Beans*. O'Reilly & Associates, 1997.
10. Extended Static Checking for Java Home Page. http://research.compaq.com/SRC/esc/.

11. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. The courier pattern. *Dr. Dobb's Journal*, Feburary 1996.
12. D. Garlan, R.T. Monroe, and D. Wile. ACME: Architectural description of component-based systems. In M. Sitaraman G.T. Leavens, editor, *Foundations of Component-Based Systems*, pages 47–68. Cambridge University Press, 2000.
13. R.M. Haefel. *Enterprise JavaBeans*. O'Reilly & Associates, 3rd edition, 2001.
14. P. Inverardi and M. Tivoli. Software architecture for correct components assembly. In *Formal Methods for the Design of Computer, Commmunication and Software Systems, Lecture Notes in Computer Science 2804*, pages 92–111. Springer, 2003.
15. J. Ivers, N. Sinha, and K.C Wallnau. A basis for composition language CL. Technical Report CMU/SEI-2002-TN-026, CMU SEI, 2002.
16. N.R. Mehta, N. Medvidovic, and S. Phadke. Towards a taxonomy of software connectors. In *Proc. 22nd International Conference on Software Engineering*, pages 178–187. ACM Press, 2000.
17. B. Meyer. Applying design by contract. *IEEE Computer*, 25(10):40–51, October 1992.
18. O. Nierstrasz, G. Arévalo, S. Ducasse, R. Wuyts, A. Black, P. Müller, C. Zeidler, T. Genssler, and R. van den Born. A component model for field devices. In *Proc. 1st International IFIP/ACM Working Conference on Component Deployment*, Berlin, Germany, 2002.
19. OMG, http://www.omg.org/cgi-bin/doc?ptc/2003-08-02. *UML 2.0 Superstructure Specification*.
20. OMG, http://www.omg.org/technology/documents/formal/components.htm. *CORBA Component Model, V3.0*, 2002.
21. J.G. Schneider and O. Nierstrasz. Components, scripts and glue. In L. Barroca, J. Hall, and P. Hall, editors, *Software Architectures – Advances and Applications*, pages 13–25. Springer-Verlag, 1999.
22. M. Shaw and D. Garlan. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice Hall, 1996.
23. Software Engineering Institute, Carnegie-Mellon University. Predictable assembly from certifiable components. http://www.sei.cmu.edu/pacc/.
24. C. Szyperski, D. Gruntz, and S. Murer. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley, second edition, 2002.
25. R. N. Taylor, N. Medvidovic, K. M. Anderson, E. J. Whitehead Jr., J. E. Robbins, K. A. Nies, P. Oreizy, and D. L. Dubrow. A component- and message-based architectural style for GUI software. *Software Engineering*, 22(6):390–406, 1996.
26. R. van Ommering, F. van der Linden, J. Kramer, and J. Magee. The Koala component model for consumer electronics software. *IEEE Computer*, pages 78–85, March 2000.