Automatic Control Flow Generation from Software Architectures

Kung-Kiu Lau and Vladyslav Ukis

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

Abstract. In a traditional software architecture, control originates in components and flows to other components via connectors. The system's control flow is fixed at design time, when components and their inter-connections are specified. Code generated from the design inherits this control flow, and consists of component code and glue code that tightly couples connected components. This means that code generated from a given software architecture is system-specific, and is therefore neither generic nor reusable. In this paper we describe an approach which allows separate reuse of component code and connector code, and thus making it possible to build architectures from pre-existing components and generic connectors. Furthermore, we show we can implement such architectures by generating control flow at run-time automatically.

1 Introduction

In a traditional software architecture [15], control originates in components (boxes) and flows to other components via connectors (lines). The system's control flow is fixed at design time, when components and their inter-connections are specified, in an Architecture Description Language (ADL), e.g. Acme [7].

Mostly ADLs do not provide any support for creating code for the system from its architecture. When they do, as in ArchJava [3,1] (based on Acme), code generated from the architecture inherits the control flow fixed at design time, and consists of component code and glue code that tightly couples connected components. This means that code generated from a given software architecture is system-specific, and is therefore neither generic nor reusable.

In this paper we describe an approach which allows separate reuse of component code and connector code, and thus making it possible to build architectures from pre-existing components and generic connectors. Furthermore, we show that we can implement such architectures by generating control flow at run-time automatically.

To achieve this we take a different approach to system construction. We take control out of components and put it into connectors. That is, in our approach, control in the system does not originate in components but in their connectors. This makes components completely encapsulated, and therefore independent and easier to reuse. Furthermore, our connectors are generic, like the Bus connector in C2 [17], and we can reuse them among different systems. To construct a system, we choose a set of pre-existing, independent components required for the system, connect them with a set of (pre-existing)

W. Löwe and M. Südholt (Eds.): SC 2006, LNCS 4089, pp. 323–338, 2006.

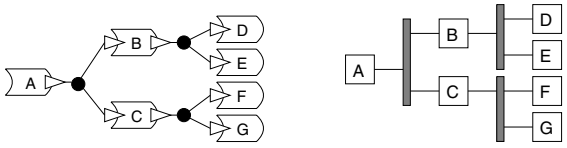
generic connectors, and generate control flow of the system automatically at run-time. Thus our approach obviates the need for generating glue code to put together components and connectors, in contrast to ArchJava. In addition, our components as well as connectors can be used in different systems with different control flows. Our automatic runtime control flow generation not only instantiates components and connectors, like the C2 Bootstrapper, but, unlike the C2 Bootstrapper, also generates the whole control flow of the system automatically at run-time.

In this paper we describe how we can generate control flow at run-time. We begin by briefly surveying current approaches for code generation from software architectures (Section 2). Next, we introduce our connectors as origins of control flow in a system (Section 3.1), and explain architectures containing our connectors (Section 3.4). Subsequently, we explain how to automatically generate control flow at runtime (Section 3.5) and provide an example (Section 4). Finally, we briefly evaluate our approach (Section 5).

2 Code Generation from Traditional Software Architectures

Among existing ADLs, Acme/ArchJava [3,1] and C2 [17] are representative examples of ADLs that support code generation from architectures.

Acme/ArchJava (Fig. 1 (a) shows an example architecture) allows automatic generation of code from an architecture. Components and connectors are generated afresh for each system. That is, neither components nor connectors pre-exist or are reused from system to system.



(a) Acme/ArchJava (b) C2

Fig. 1. Software architecture examples

In C2 (Fig. 1 (b) shows an example architecture) components have to be coded first. Components communicate by sending events to each other. Their code must explicitly identify events they can deal with and provide corresponding actions. Events are transported by buses between components. The bus is a generic connector, and is not generated afresh for every system but is reused in all systems. To implement an architecture, C2 provides a Bootstrapper, which allows instantiation of components and connectors at run-time. These instances together with event-handling constitute the runtime system. Thus in C2 the bus is reused but not the components, because the latter are 'hard-wired' to events for a specific system.

No ADL inherently intends both component and connector reuse from an architecture description. This is exactly what our approach endeavours to achieve. We want to have pre-existing components as well as pre-existing connectors, and reuse them to build many architectures.

In the ArchJava example in Fig. 1 (a) for instance, component A knows connector AB. The connector AB in turn knows component B. Thus, A cannot be reused independently without AB and AB without B. Furthermore, component B knows connector BD. The connector BD in turn knows component D. Thus, B cannot be reused without BD and BD without D. In other words, neither components nor connectors in ArchJava are independently reusable entities.

In the C2 example in Fig. 1 (b), connectors AB and BD etc. are constructed from one generic bus connector template and, unlike in ArchJava, are not coded afresh for each component connection. Thus connectors AB, BD etc. are generic and independently reusable. However, component A sends a specific event with a specific format, say AB_Event, to the connector AB. The connector AB dispatches the AB_Event to component B. Component B is waiting for the arrival of this specific event, knows its format and how to handle it. Moreover, once component B has processed the AB_Event, it originates another event, say BD_Event, to the bus connector BD. The bus connector BD dispatches the BD_Event to component D. Component D is waiting for the arrival of the BD_Event, knows its format and how to handle it. In other words, components in C2 wait for specific incoming events from and send (or originate) specific outgoing events to other components. Therefore, components in C2 are not independently reusable encapsulated entities.

By contrast, we want to be able to reuse components A, B, D etc. independently as well as connectors AB, BD etc.

3 Our Approach

In this section we explain our approach. The key characteristics of our approach are that (i) components pre-exist and are reusable; (ii) connectors (pre-exist and) are generic and reusable; (iii) run-time systems can be generated from architectures by automatically generating control flow.

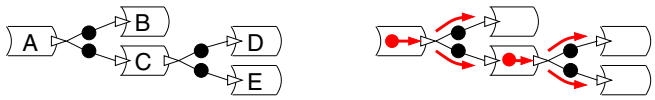
To make our components reusable, we make them encapsulated and thus independent, by taking control out of them. Thus in our approach, components are units of computation (linked by connectors). A component is a unit of software with (i) an *interface* that specifies the services it provides (i.e. its methods) 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 [16]. However, our components do not invoke methods or services in other components. Rather, they only perform their provided services (methods) when they are invoked from outside, by connectors. Thus our components encapsulate computation.

We put control in connectors. Connectors are composition operators that compose components into systems. They are *exogenous*, i.e. 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 in general, and control in particular. Exogenous connectors play a fundamental role on our approach.

3.1 Exogenous Connectors

Exogenous connectors were introduced in [12]. Here, we briefly explain them.

The distinguishing characteristic of exogenous connectors is that they encapsulate control. In traditional ADLs, components are supposed to represent *computation*, and



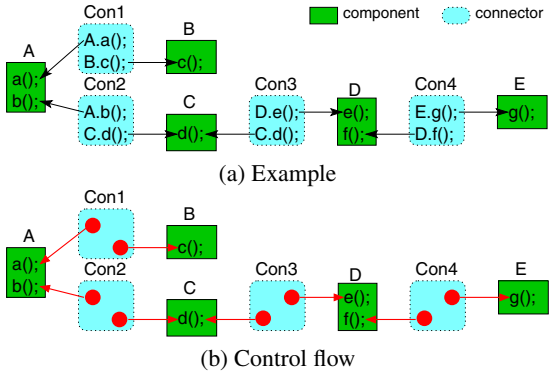
(a) Components and connectors (b) Control flow

Fig. 2. Traditional ADLs

connectors *interaction* between components [13] (Fig. 2 (a)). Actually, however, components represent computation as well as *control*, since control originates in components, and is passed on by connectors to other components. This is illustrated by Fig. 2 (b), where the origin of control is denoted by a dot in a component, and the flow of control is denoted by arrows emanating from the dot and arrows following connectors.

In this situation, components are not truly independent, i.e. they are tightly coupled, albeit only indirectly via their ports, and the control flow between components is fixed at their design time.

By contrast, in exogenous connection, control originates in and flows from connectors, leaving components to encapsulate only computation. This is illustrated by Fig. 3.



(a) Example

(b) Control flow

Fig. 3. Connection by exogenous connectors

In Fig. 3 (a), components do not call methods in other components. Instead, all method calls are initiated and coordinated by exogenous connectors. The latter's distinguishing feature of control encapsulation is clearly illustrated by Fig. 3 (b), in clear contrast to Fig. 2 (b).

Exogenous connectors thus encapsulate control (and data), i.e. they *initiate* and *coordinate* control (and data). With exogenous connection, components are truly independent and decoupled.

Exogenous connection is not provided by any existing ADLs. However, exogenous connection has been defined as exogenous coordination in coordination languages for concurrent computation [2]. Also, in object-oriented programming, the courier pattern [6] 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.

3.2 Connector Type Hierarchy

The concept of exogenous connection entails a type hierarchy of exogenous connectors. Because they encapsulate all the control in a system, such connectors have to connect to one another (as well as components) in order to build up a complete control structure for the system. For this to be possible, there must be a type hierarchy for these connectors.

In the connector type hierarchy for our approach, components are obviously a basic type. Because 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.

Example 1. (Exogenous Connector Hierarchy). Consider a system whose architecture can be described in Acme [7] and C2 [17] as in Fig. 1 (a) and (b) respectively. Using exogenous connectors in our approach, the corresponding architecture is that shown in Fig. 4.

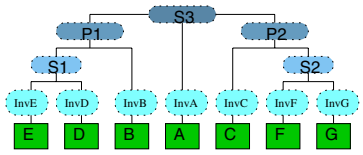


Fig. 4. Exogenous connection example

At the lowest level, level 1, we use *invocation* connectors that connect to individual components and make calls into them. There are no other kinds of connectors at this level. In Fig. 4, the invocation connectors are InvE, InvD, InvB etc.

At the next level, level 2, we need a *selector* connector to implement branching in the system. Such a connector connects connectors and makes a call into a selected one of the connectors. In Fig. 4, at level 2, selector S1 connects InvE and InvD, and decides whether to call E or D depending on the selection condition it receives from its parent connector P1. Similarly, S2 connects and selects from F and G.

At level 3, we need a *pipe* connector to implement sequential control. Such a connector connects connectors and makes consecutive calls into these connectors in the

order in which they are connected to it. In Fig. 4, P1 is a pipe connector. It connects to InvB (which calls a method in B) and passes the result to the selector S1. S1 uses the result as a selection condition to select component E or D. Similarly, the pipe P2 effects sequential control between C and selector S2.

Finally, at level 4, the top level, there is only one connector. This is a selector S3, which selects P1 or P2 depending on the top-level (user) input.

In general, connectors at any level other than the first can be of variable arities; connectors at any level higher than 2 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 $(n - 1)$. At the top level, there is always just one connector. A detailed definition of the hierarchy can be found in [12,11].

3.3 Implementing Generic Connectors

Exogenous connectors can be implemented as generic connectors, such that: (i) generic connector templates can be defined and stored; (ii) these connector templates can be deployed to a system; and (iii) connector instances can be created and used to build the control structure of any specified system (with exogenous connectors). In particular, we want to do so for *any* connector at *any* level. In [12] we show an implementation in Java that is generic only in the sense of (i), and that only defines connectors for *specific* levels. Here we describe how we can define connectors at *any* level that are generic in the sense of (i), (ii) and (iii). We use C# in .NET for the implementation.

We implement three kinds of connectors (*invocation*, *pipe* and *selector*) as a hierarchy of classes, with a base class *Connector*.

The *Connector* class has several *Execute* methods for executing either a single given method (with its parameters) or a given set of methods (with their parameters). These are the following `public virtual void` methods:

```
... Execute (string method, object[] params);//(1)
... Execute (string[] methods, object[] params);//(2)
... Execute (int cond, string method, object[] params);//(3)
... Execute (int cond, string[] methods, object[] params);//(4)
```

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).

Only the invocation connector makes calls into components from within its *Execute* method (1).

The selector connector's *Execute* method can be passed a list of methods (4). Consider the case of just one method (3). In this case, the *Execute* method of a selector connector is used for calling one method on the connector inside the selector which gets selected according to the condition *cond* which is passed into the method. In our current implementation, the selection condition is an integer but it can easily be extended to other types in future.

The selector assumes that all the connectors in it can in principle deal with the method passed into it. Therefore it is also sufficient to provide only one list of

parameters. Whichever connector gets selected, the method *method* and parameters *params* will be passed to it.

The *Execute* method of a pipe connector (2) is represented by a loop, which sequentially processes all the connectors in it. Basically, the pipe connector takes the first connector, makes a call into it, obtains the result and makes a call into the second connector passing the result obtained from the first connector as a parameter into the second one and so on until the end of the loop is reached.

In the loop the first thing is to check whether we are at the beginning of the loop. If we are, then the parameters passed into the *Execute* method can be used as they are, to be passed into the first connector. On the other hand, if we are in the middle of the loop, the parameters to be passed on to the next connector are the results from the previous one.

Next if the connector to be called in the current loop iteration of the pipe is a selector connector, we have to extract the first parameter from the *Execute* method's parameter list if we are at the beginning of the loop, or the first element of the result array from the previous invocation if we are in the middle of the loop, and pass it to the selector connector as a condition.

Then if we are at the first loop iteration we can call into the selector straight away, but otherwise we have to adjust the method array and remove the first element from it because the first method has already been processed in the previous loop iteration.

If the connector in the current loop iteration is not a selector, we do not have to bother with the first element in the parameter list to be processed as a selection condition, and can call the *Execute* method straight away considering the necessary method array adjustment for each loop iteration.

Eventually the Result is retrieved from the connector processed in the current iteration, and will be used in the next iteration as parameter list for the next connector in the pipe. Once the end of the loop is reached, the Result is returned by the pipe.

The connectors we present here are generic because they are independent, self-contained and can be used by any application. As shown in the above description, no application-specific logic has been put into the connectors. In fact, in a general sense they could even be thought of as light-weight components in the system.

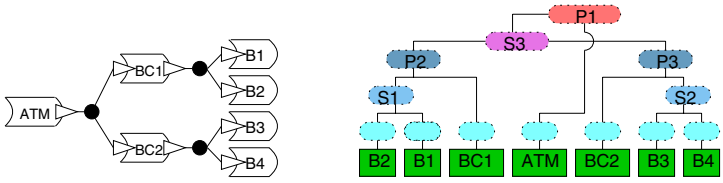
Exogenous connectors form a hierarchy and thus can contain one another. Thus, pipe and selector connector can contain invocation, pipe or selector connectors. It is possible to add a connector to the "host" connector after the "host" connector has been created when building a connector hierarchy. This allows for "late-binding" of connectors, which is used for system control flow generation.

3.4 Architectures with Exogenous Connectors

Having implemented generic exogenous connectors, in this section we show how architectures can be defined using them. Just as exogenous connection entails a connector type hierarchy, so the latter in turn entails a strictly hierarchical way of constructing systems by composing components. As illustrated by Figure 4, in such a system, components form a flat layer, and the entire control structure (of connectors) sits on top of this. Beyond level 1, the precise choice of connectors, the number of levels of connectors, and the connection structure, depend on the relationship between the behaviour

of the individual components and the behaviour that the whole system is supposed to achieve. Whatever the control structure, however, it is strictly hierarchical, which means that there is always only one connector at the top level. This is the connector that initiates control flow in the whole system.

Example 2. (The Bank Example). Consider a bank system, whose architecture is described in Acme in Figure 5 (a). The system has just one ATM that serves two bank



(a) Acme (b) Exogenous connection

Fig. 5. Architecture of the bank example

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.

At level 1, each component has an invocation connector. At level 2, 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-2 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 3, we have two pipe connectors $P2$ and $P3$, for $BC1$ and $BC2$ respectively. At level 4, $S3$ is a selector connector that selects the customer's bank consortium from consortia $BC1$ and $BC2$. Finally, at level 5, the top level, the pipe connector $P1$ initiates the bank system's operational cycle by passing customer requests and card information to the ATM, invoking the ATM's methods, and then passing the resulting value to connector $S3$.

3.5 Automatic Control Flow Generation

Separation of control flow and computation using exogenous connectors means that control flow is not kept inside components like in current ADLs but can be managed outside. Having implemented generic exogenous connectors, in this section we show how a system's control flow can be generated automatically, given its architecture, i.e. the connection structure for the components.

As depicted in Figure 6 (which should be read from bottom to top, as indicated by the arrow on the left), to generate a system's control flow we need 3 kinds of entities: (a) independent components; (b) generic exogenous connectors; and (c) an XML description of the system's architecture, i.e. the connection structure of the system. These

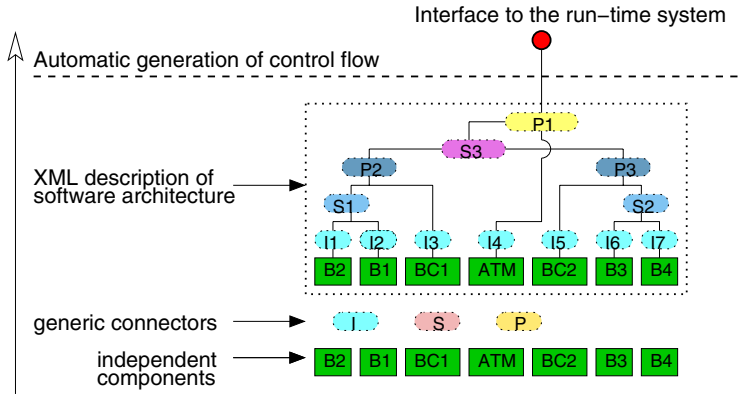


Fig. 6. Automated control flow generation

3 entities are independent from one other, i.e. components can be connected by any connectors depending on a specific system's needs, and connectors can take part in any connection structure.

The output of the control flow generation is a run-time system constructed in accordance with the given connection structure description, along with an interface, which is the top-level connector in the architecture. The system constructed provides all control flow paths possible in the system specified by (c). A particular run-time request to the system may not use every control flow path available. Nevertheless, the system construction ensures that all possible control flow paths are available to serve all requests placed on the system through the top-level connector in the architecture.

Application-*independent* templates for connectors can be created as shown in Section 3.3 and reused for different applications by creating application-specific instances. Note that connector template instances are not ordinary class instances in the sense of object-oriented programming. When a connector template is instantiated it gets adapted to the current place in the connection structure. The generic exogenous connectors can be deposited in a repository and retrieved on demand for each application. Furthermore, for any specific application with an exogenous control or connection structure, the generic connectors can be instantiated, on the fly, into the instances in the latter's connection structure. This means that it is possible to generate the control flow of a system dynamically and automatically from its architecture.

To illustrate this, consider the connection structure of the Bank example in Figure 6. The system contains three pipe connectors and three selector connectors (as well as seven invocation connectors). Each of these connectors hosts different connector types (and in different numbers). For example, the pipe $P1$ hosts a selector $S3$ and an invocation connector $I4$ for the component *ATM*, whereas the pipe $P2$ hosts a selector $S1$ and an invocation connector $I3$ for the component $BC1$. Although the two pipes are doing completely different things, they have been constructed from the same template. The template is generic enough to embody different instances. So, $P1$ is an instance of the pipe template that hosts the selector $S3$ and the invocation connector $I4$, and $P2$ is an instance that hosts the selector $S1$ and the invocation connector $I3$.

The same applies to selector and invocation connectors (and indeed to any connector). A selector connector template can take any number of any connectors, and an invocation connector template can call any method on any component.

Thus we can automate the process of control flow construction for any system with an exogenous connection structure by instantiating connector templates into instances in the latter.

Note that, by contrast, ADL systems do not have these properties. In such systems, connectors are not generic but system-specific, and components, rather than connectors, form a hierarchy. Only C2 makes use of a generic (bus) connector. However, in C2 components originate control to other components and therefore cannot be reused independently as self-contained units of computation.The chain of dependent components is laid down at components' design time. By contrast, we do it at run-time.

3.6 Connection Structure Description

In order to build up a control structure on the fly, it needs to process a system's connection description. We choose to write the description in XML because: (a) XML itself is hierarchical, and so is particularly suited to expressing our connector hierarchies; (b) the system description can be automatically checked against a pre-defined XML schema, thus eliminating (some) errors right at the beginning; (c) there is good tool support for XML, e.g. we use XMLSpy from Altova; (d) the system integrator can be guided by a tool while developing a system control structure description according to the XML schema; (e) XML schemas are extensible in a consistent manner [5]; this is important because when the schema is extended to include new connector types, for instance, old system descriptions, which have been checked against the old schema, will be able to pass the schema check using the new schema. Using XML for system description is also favoured by XML ADLs [14,8,4].

The XML schema we use for system control structure description is depicted in Figure 7. The top-level XML element is called "ExADL" and has two child elements: (i) connector_types and (ii) system, in that order. (i) contains an extensible specification of exogenous connector types which are generic and not system-specific; whilst (ii) contains a (system-specific) specification of the system using these connector types. Connector types presented here include invocation, pipe and selector connectors.

A system can contain any number of connector types which can contain one another. The connector type hierarchy defined in the schema is of course the same one that we used for implementing these connectors.

Note that connector types presented here are not the only ones possible. We show only these connector types here because they are used in the Bank Example. In general, any exogenous connector types are conceivable. For example, a repeater connector, which repeats some invocations into a component, or a sequencer connector, which has the semantics of the pipe connector but does not pipe values from one component to another one. What is important is that all these connectors can be described using system control structure description and instantiated at runtime. That is, the infrastructure for building systems using exogenous connectors is defined by the extensible XML schema for system control structure description.

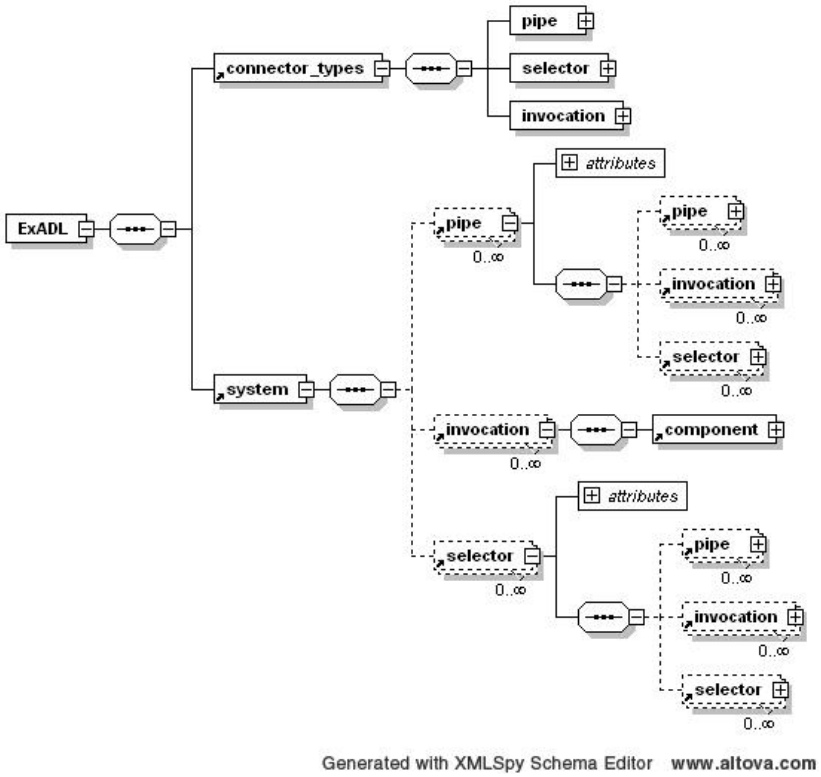


Fig. 7. XML schema for system connection structure description

As an example of system connection structure description, the bank system can be described by the outline in Figure 8.This can be read as: 'A pipe $P1$ contains an invocation connector and a selector $S3$. The invocation connector contains a component ATM. The selector $S3$ contains a pipe $P2$, which contains a component $BC1$, and so on'.

3.7 Implementation of Control Flow Generation

To generate a system's control flow, its XML description is processed. First of all the XML description is checked against the schema shown in Figure 7. If the XML system description does not pass the schema check, the system will not be created. This enforces the connector hierarchy to be always well-defined by the schema. During the processing of the system element, the connector types are retrieved first and stored for future use. A connector type is instantiated each time a specific connector occurs in the system connection structure description. For example, each time a pipe element occurs in the XML description of the system, an instance of a pipe is created from the information stored before.

To describe the implementation, we follow the sequence of operations that are carried out to process a system control structure description. First, the system control flow

```
<system>
  <pipe name="P1">
    <invocation>
      <component name="ATM" type="Components.ATM, Components"/>
    </invocation>
    <selector name="S3">
      <pipe name="P2">
        <invocation>
          <component name="BC1" type="Components.BankConsortium ...
        </invocation>
        <selector name="S1">
          <invocation>
            <component name="B1" type="Components.Bank ...
          </invocation>
          <invocation>
            <component name="B2" type="Components.Bank ...
          </invocation>
        </selector>
      </pipe>
      ...
```

Fig. 8. Connection structure description for the bank example

description gets validated against the XML schema and gets loaded unless the description violates the schema. Second, information about the location of each connector class is stored for creating connector instances in future. We use XPath expressions to retrieve the XML nodes (e.g. "//connector_types/pipe"). The information stored is a piece of text containing the class name and a .NET assembly name containing the class. Using this information .NET runtime (CLR) can load the assembly into a process and create an instance of the class inside. Third, the top-level connector is identified and created, Then system control flow construction begins. The complete system is created beneath the top-level connector, using a recursive method:

```
private void LoadSystem(XmlNode theXmlNode,
                        Connector theCurrentConnector) {...}
```

This recursive method has 2 parameters: (i) the current XML node in the system control structure description to be processed; and (ii) the current connector, which will take the connectors created from the child nodes of the XML node passed into the method as child connectors. Thus when entering the method we always have a connector created in the previous iteration and its XML representation. The method iterates through the child nodes of that node, creates connectors out of them and puts each of these connectors as a child connector into the connector passed into the method.

The recursion itself can only occur when processing either a pipe or a selector connector. An invocation connector cannot cause the recursion since the only XML node that can be beneath invocation is component, according to the XML schema. On the other hand, we do not know which XML node will occur after pipe or selector. The schema only enforces that it will be either pipe, selector or invocation. In order to investigate what is below a pipe or a selector we engage in a recursion passing the necessary

parameters, namely the current connector and its XML representation, and in the next iteration explore the child nodes. The recursion ends when an invocation connector is found.

During the construction of the system control flow all possible control flow paths in the system are laid down, while a particular request to the system does not necessarily makes use of all of them but follows some paths necessary to answer the request.

4 Example

Now we illustrate the use of exogenous connectors for automatic runtime control flow generation, using the bank example (Example 2), with the architecture described in Figure 5 (b).

The first step is to implement the components. In our implementation, components are C# classes with public methods (that can be invoked by the invocation connectors) for the usual ATM operations like insert card, enter password, withdraw, deposit, check balance, etc. The objects (of these classes) do not call methods in other components.

The second step is to specify the system in XML following the XML schema. We have already done this in Figure 8.

The third step is to actually construct the system according to the process outlined above. The result is the running system with control flow as shown in Figure 6.

Now we briefly explain how the automatically generated bank system works, and therefore how it can be used to provide services, by means of an example. Consider the service request of getting the balance of an account. The get balance operation (illustrated for card 4711) is implemented by using *TopLevelConnector* of the bank system, as follows:

```
TopLevelConnector.Execute(new string[] {"GetBankConsortiumID_",
 "GetBranch_", "GetBalance"}, new object[] {4711});
```

The top-level connector $P1$ gets a list of methods, namely *GetBankConsortiumID_*, *GetBranch_* and *GetBalance*, and parameters to be propagated through the system. Only invocation connectors in ATM, $BC1$ and $B1$ respectively call these methods. The connectors themselves draw on various *Execute* methods offered by their base class *Connector* to propagate the necessary information down towards invocation connectors. Where the control flow can pass (at which connector and component) was specified before in the system description. The concrete control flow for a request depends on request parameters. For example, a particular bank is selected for executing an operation on an account according to the account number of the customer.

For the get balance operation, the control flow involved is shown in Figure 9.

Note that the control flow for get balance operation does not use all possible control flow paths laid down on system construction but rather uses a part of them. Figure 6 shows that the system contains all the possible control flow paths. Figure 9 depicts control flow paths necessary for serving the request to get an account balance. Another request may need completely different paths than those used when serving account balance request.

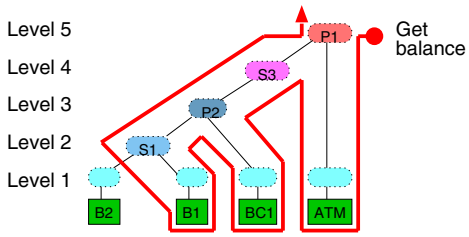


Fig. 9. Control flow for get balance

Other operations to be performed by the Bank System like deposit and withdraw can be implemented as follows:

Deposit $100 onto account the card 4711 belongs to:

```
TopLevelConnector.Execute(new string[]
 {"GetBankConsortiumID", "GetBranch", "Deposit"},
 new object[] {"100", "4711"});
```

Withdraw $100 from account the card 4711 belongs to:

```
TopLevelConnector.Execute(new string[]
 {"GetBankConsortiumID", "GetBranch", "Withdraw"},
 new object[] {"100", "4711"});
```

Besides the Bank Example we have implemented a complex Automated Train Protection System (ATP) using exogenous connectors. In that system we implemented some other connectors in addition to those presented in this paper and we could reuse connectors from this paper in the ATP system. For lack of space we do not discuss the ATP System here.

5 Discussion and Concluding Remarks

In this paper we have presented an approach to automatic runtime system control flow generation from software architectures using exogenous connectors. In particular, we showed our procedure for control flow construction. As far as we know, our approach is unique because it generates control flow of systems consisting of independent, reusable components automatically.

Code generators like the one in ArchJava generate code with components originating control flow to other components. Tools like Bootstrapper in C2 do not create control flow of the system at runtime but only instantiate components and connectors, with control flow already implemented in components and via connectors. In other words, traditional ADLs do not allow automatic runtime control flow generation for a system.

Furthermore, ADLs do not have generic and hierarchical connectors. XML-based ADLs like xADL 1.1 [8] and xADL 2.0 [4], which have XML descriptions of their architectures, do not generate control flow automatically at runtime.

Table 1 summarises related approaches and shows the differences to our proposed approach.

Table 1. Comparison with related architectures

Approach	Access to component	Control origin	Component reuse	Connector reuse	Automated control flow generation
ArchJava/ACME	by method call	component	no	no	no
xADL	by method call	component	no	no	no
C2	by event	component	no	yes	no
Exogenous	by method call	connector	yes	yes	yes

Our future work is concerned with predictability of system properties resulting from composing components using connectors into a system. We have shown that automatic composition is possible by constructing system's control flow on the fly. However, it is highly desirable as well to be able to predict the result of this automated control flow construction before it actually takes place. Therefore we are working on Deployment Contracts [10] for components, which is metadata [9] attached to the components, with a view to being able to analyse that metadata before the actual composition takes place. The analysis should flag incompatible components for composition. By having this, we will be able to predict conflicts by doing some compositional reasoning.

References

1. J. Aldrich, C. Chambers, and D. Notkin. ArchJava: Connecting software architecture to implimentation. In *Proc. ICSE 2002*, pages 187–197. IEEE, 2002.
2. 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.
3. ArchJava web page. `http://archjava.fluid.cs.cmu.edu/index.html`.
4. E.M. Dashofy, A. van der Hoek, and R.N. Taylor. A highly-extensible, XML-based architecture description language. In *Proc. Working IEEE/IFIP Conference on Software Architecture*, pages 103–112. IEEE Computer Society, 2001.
5. L. Dykes, E. Tittel, and C. Valentine. *XML Schemas*. Sybex Inc, 2002.
6. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. The courier pattern. *Dr. Dobb's Journal*, Feburary 1996.
7. D. Garlan, R.T. Monroe, and D. Wile. Acme: Architectural description of component-based systems. In G.T. Leavens and M. Sitaraman, editors, *Foundations of Component-Based Systems*, pages 47–68. Cambridge University Press, 2000.
8. R. Khare, M. Guntersdorfer, P. Oreizy, N. Medvidovic, and R. N. Taylor. xADL: Enabling architecure-centric tool integration with XML. In *Proc. 34th Hawaii Int. Conf. on System Sciences*, 2001.
9. K.-K. Lau and V. Ukis. Component metadata in component-based software development: A survey. Preprint 34, School of Computer Science, The University of Manchester, Manchester, M13 9PL, UK, October 2005.
10. K.-K. Lau and V. Ukis. Deployment contracts for software components. Preprint 36, School of Computer Science, The University of Manchester, Manchester, M13 9PL, UK, February 2006.

11. K.-K. Lau, V. Ukis, P. Velasco, and Z. Wang. A component model for separation of control flow from computation in component–based systems. In *Proceedings of the 1st International Workshop on Aspect-Based and Model-Based Separation of Concerns in Software Systems, ENTCS, www.elsevier.nl/locate/entcs*, Nuremberg, Germany, November 2005.
12. K.-K. Lau, P. Velasco Elizondo, and Z. Wang. Exogenous connectors for software components. In *Proc. 8th Int. SIGSOFT Symp. on Component-based Software Engineering, LNCS 3489*, pages 90–106, 2005.
13. 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.
14. S. Pruitt, D. Stuart, W. Sull, and T.W. Cook. The merit of XML as an architecture description language meta-language. Microelectronics and Computer Technology Corporation, 1998.
15. M. Shaw and D. Garlan. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice Hall, 1996.
16. C. Szyperski, D. Gruntz, and S. Murer. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley, second edition, 2002.
17. 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.