August 2005

A Container for Automatic System Control Flow Generation using Exogenous Connectors

Kung-Kiu Lau and Vladyslav Ukis

School of Computer Science
The University of Manchester
Preprint Series
CSPP-31

A Container for Automatic System Control Flow Generation using Exogenous Connectors

Kung-Kiu Lau and Vladyslav Ukis

August 2005

Abstract

Today's component models like EJB and CCM use containers to instantiate and manage components. A component in these component models can originate control flow to other components to fulfil its tasks, and control flow of the system is laid down at design time of its components. The container can instantiate the system, given a set of such components. As components originate control flow to each other, their reuse potential is limited because they cannot be reused independently but need some other components they originate control to. The container does not take part in the system's control flow construction at all, leaving glue code for component connection in components themselves. In this report, we present a container which can automatically construct control flow for a system consisting of independent components not calling each other and exogenous connectors to connect them. As the components do not call each other their reuse potential is much higher than that of current component models. The container, unlike current containers, not only instantiates the components but is also in charge of control flow construction of the component-based system.

Keywords: Exogenous connectors, component composition, container, .NET

Contents

List of Figures

List of Tables

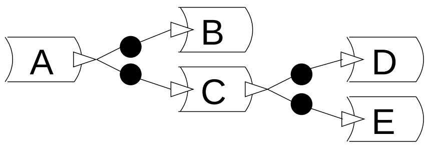
1 Introduction

In component-based software development [28], composition is a central issue. A key challenge is to design composition operators that are not only practical for component disentanglement, but are also amenable to compositional reasoning, since the nature of components demands reasoning about component behaviour both statically and dynamically, i.e. at build time as well as at run time. Architecture description languages (ADLs) [27] provide connectors as composition operators, and architectural reasoning has recently begun to be investigated, e.g. [1]. However, traditional ADLs do not separate computation (components) from interaction (connectors) as cleanly as intended, thus complicating architectural reasoning. Components not only perform computation, but also initiate control, which is then passed by the connectors to other components. To make compositional reasoning more tractable, we believe it is necessary to improve encapsulation of computation (components) as well as control (connectors). To this end, we have proposed exogenous connectors. These connectors provide composition mechanisms different from those in existing component models (including ADLs) [18, 19], in that they completely capture control, leaving components to encapsulate only computation. In this report, we show that exogenous connectors allow for automatic control flow construction of a system. To be useful in practice, connectors need to be generic, i.e. they should be not only instantiable, but also reusable for different applications. ADL connectors are not generic in this sense, since they are defined and created afresh for each application. Generic connectors enable users to not only construct systems more easily but also to write (and therefore maintain) less code for each application. We will show how exogenous connectors can be implemented in a generic way, and how a container can be implemented that can use (generic) exogenous connectors to automatically compose software components.

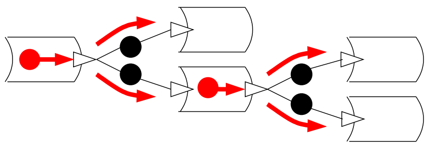
2 Exogenous Connectors

In [17], we present exogenous connectors for software components. In this section, we briefly explain these connectors in the context of related work.

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



(a) Components and connectors



(b) Control flow

Figure 1: Traditional ADLs.

teraction between components [21] (Figure 1 (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 Figure 1 (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.

In general, component connection schemes in current component models (including ADLs) use message passing, and fall into two main categories: (i) connection by direct message passing;

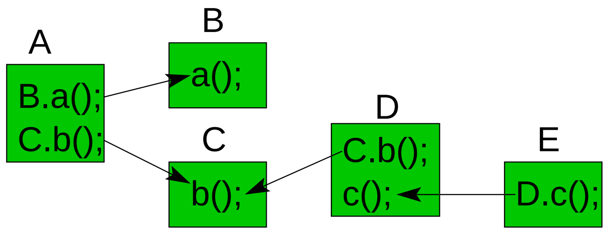


Figure 2: Connection by direct message passing.

and (ii) connection by indirect message passing. Direct message passing corresponds to direct method calls, as exemplified by objects calling methods in other objects (Figure 2), using method or event delegation, or remote procedure call (RPC). Software component models that adopt direct message passing schemes as composition operators are Enterprise JavaBeans [14], CORBA Component Model [25], COM [6], UML2.0 [24] and KobrA [5]. In these models, there is no explicit code for connectors, since messages are 'hard-wired' into the components, and so connectors are not separate entities.

Indirect message passing corresponds to coordination (e.g. RPC) via connectors, as exemplified by ADLs. 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

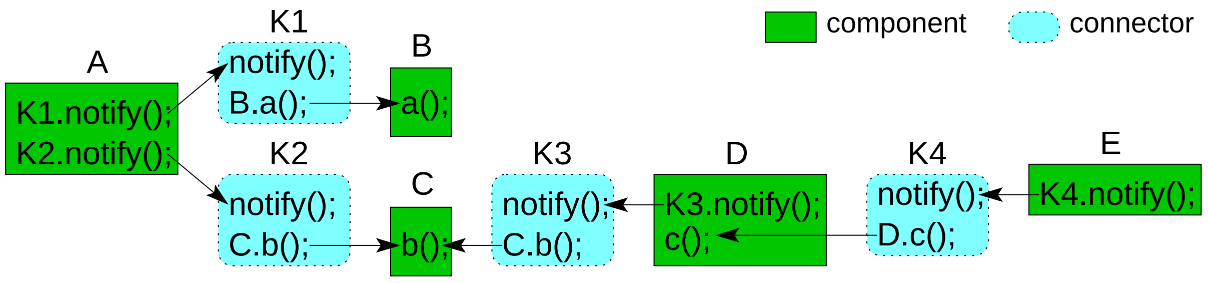


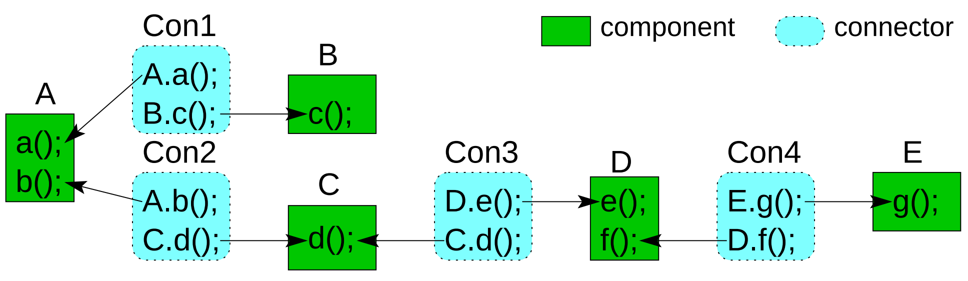
Figure 3: Connection by indirect message passing.

component to another component we use a connector that when notified by the former invokes a method in the latter (Figure 3). Besides ADLs, other software component models that adopt indirect message passing schemes are JavaBeans [10], Koala [30], SOFA [26], PECOS [23], PIN [15] and Fractal [7].

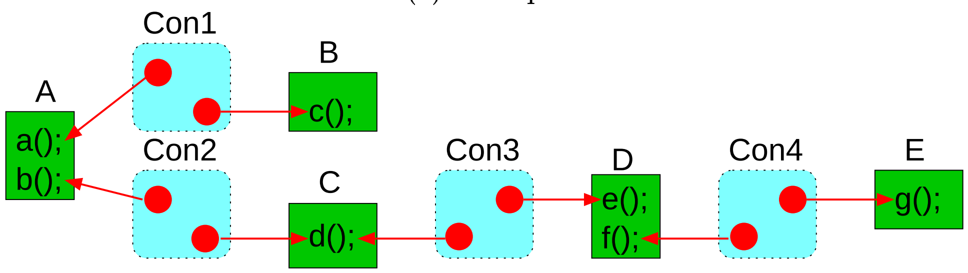
In connection schemes by message passing, direct or indirect, control originates in and flows from components, as in Figure 1 (b). This is clearly the case in both Figure 2 and Figure 3.

By contrast, in exogenous connection, control originates in and flows from connectors, leaving components to encapsulate only computation. This is illustrated by Figure 4. In Figure 4 (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 Figure 4 (b), in clear contrast to Figure 1 (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



(a) Example



(b) Control flow

Figure 4: Connection by exogenous connectors.

decoupled.

Exogenous connection is not provided by any existing software component models (including ADLs). However, exogenous connection has been defined as exogenous coordination in coordination languages for concurrent computation [3]. Also, in object-oriented programming, the courier pattern [12] 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.

2.1 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. Therefore such a hierarchy must be defined for any component model that is based on exogenous connection. We are developing such a component model, and in this section we describe the connector type hierarchy for our model.

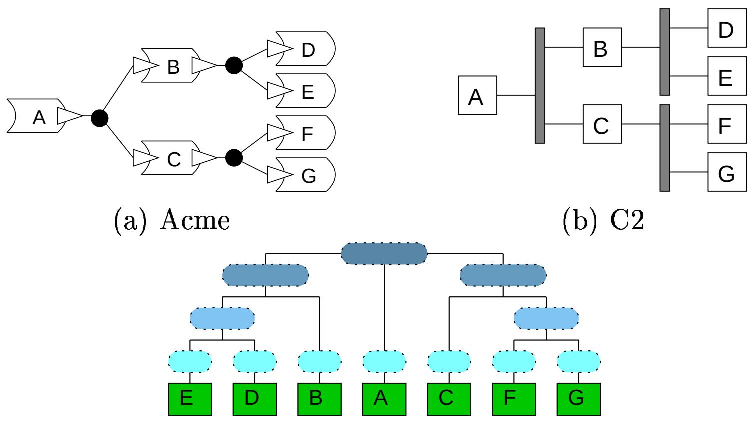
In our component model,[1] components are units of computation linked by exogenous connectors. A component is a unit of software with (i) an *interface* that specifies the services it provides (i.e. its methods) and the services it requires, and the dependencies between the two sets of services; and (ii) *code* that implements the provided services. In essence it is similar to Szyperski's definition [28]. 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.

[1]We do not give a full description; it is not necessary here.

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.

In the connector type hierarchy for our component model, 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.



(a) Acme (b) C2

(c) Exogenous connection

Figure 5: Corresponding architectures.

For example, consider a system whose architecture can be described in the Acme [13] and C2 [29] ADLs by the architectures in Figure 5 (a) and (b) respectively. Using exogenous connectors in our component model, the corresponding architecture is that shown in Figure 5 (c). In the latter, 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 connectors at levels 3 and 4 are of variable arities and types. Note that at the top level, there is only one connector.

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 $(n-1)$.

The complete connector type hierarchy (omitting methods and their parameters) is shown in Figure 6. Note that level-1 and level-2 connectors are not polymorphic, but connectors at higher levels are.

Basic types:	Component, Result;	
Connector types:		
$L1 \equiv$ Invocation	$\equiv$	Component $\longrightarrow$ Result;
$L2$	$\equiv$	$L1 \times \ldots \times L1 \longrightarrow$ Result;
$L3$	$\equiv$	$L \times \ldots \times L \longrightarrow$ Result where L is either $L1$ or $L2$;
...		

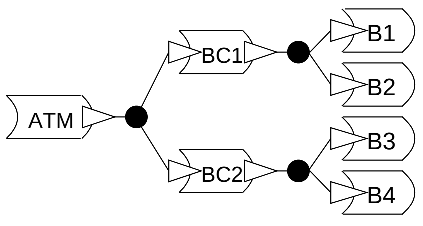
Figure 6: Connector type hierarchy.

2.2 Component Composition

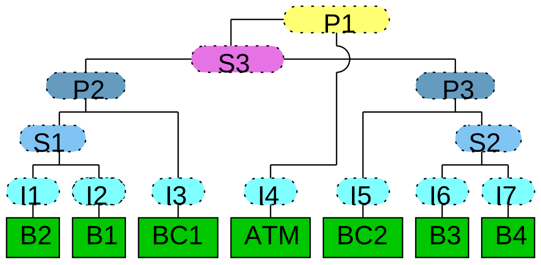
Just as exogenous connection entails a connector type hierarchy, so the latter in turn entails a strictly hierarchical way of constructing systems by composing components. As illustrated by Figure 5 (c), in such a system, components form a flat layer, and the entire control structure (of connectors) 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.

2.2.1 The Bank Example

Consider a bank system, whose architecture is described in Acme in Figure 7 (a). The system has



(a) Acme



(b) Exogenous connection

Figure 7: Architecture of the bank example.

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.

In [17] we implement the bank system using exogenous connectors and the architecture in Figure 7 (b) (a refinement of Figure 5 (c)). 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 Automated Composition

In [17], we demonstrate the feasibility of implementing the hierarchy of exogenous connectors in Figure 6, and using them to construct component-based systems. As an illustration, we *manually* construct the bank system in Figure 7 (b) using exogenous connectors. In this report, we go a step further and show that using exogenous connectors we can automate component composition. As an illustration, we will show that the bank system can be composed *automatically* by using exogenous connectors.

The key properties of these connectors that make this possible are *separation of control flow and computation*, *genericity* and *hierarchy*. Separation of control flow and computation gives us an additional degree of freedom in the system, which is not kept inside components like in current component models but can be governed by a container. Genericity means connectors can be implemented as application-independent templates, from which instances can be created for specific applications. This is in contrast to connectors in current ADLs, whose components expose methods rather than events. Systems employing event dispatching as communication means of components, e.g. C2 ADL, have buses which are generic connectors. They do not possess any application-specific code and can be used for dispatching of any events. C2 components are nonetheless dependent on each other as they originate control to each other. Systems employing method calls as communication means of components do not possess generic connectors. Their code must be generated for each application. In Acme/ArchJava [4, 2], for instance, when a system description is compiled into Java, the resulting code for the connectors is specific to that system. This code cannot be (re)used for other systems, and the system developer has to maintain it just for this system.

By contrast, exogenous connectors can be implemented in a generic manner, so that different instances can be created from the same template. Furthermore, these instances can be created automatically by a generic container from the system description for the particular application. Consequently, from the system developer's point of view, creating a system requires only giving the container the system description (together with code for the components).

What makes the generic container possible is the use of generic exogenous connectors, and their type hierarchy, which enforces a strictly hierarchical way of building the control structures of component-based systems, as we have seen in the bank example in Section 2.2.1.

In this section, we describe an implementation of exogenous connectors together with a generic container which can compose connectors and hence systems automatically.

3.1 Generic Exogenous Connectors

Now we show how we implement exogenous connectors in a generic way such that: (i) in the design phase, generic connector templates can be defined and stored; (ii) in the deployment phase, these connector templates can be deployed in the container; and (iii) in the runtime phase, connector instances can be created (automatically) by the container 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 [17] 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). For reasons that will become apparently later, we use C# in .NET for the implementation.

We implement three kinds of connectors as a hierarchy of classes, with a base class *Connector* (Figure 8).

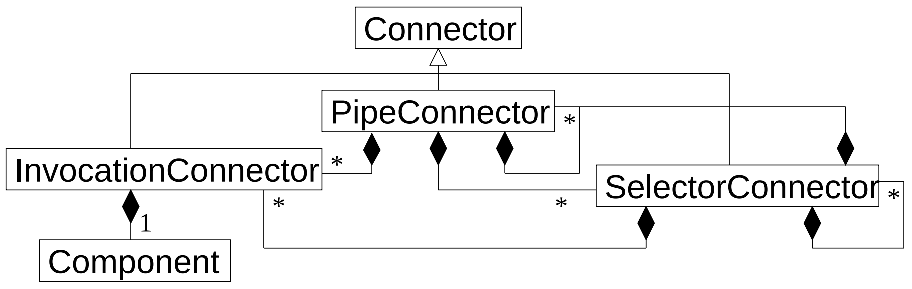


Figure 8: Class hierarchy of exogenous connectors.

The *invocation* connector connects to a component and makes calls into it. It can only appear at the bottom of the connector hierarchy, i.e. level 1.

The *pipe* connector connects either invocation connectors or other pipe or selector connectors. It makes consecutive calls into connectors it connects in the order in which they are connected to it. It can appear at any level greater than 1 in the connector hierarchy.

The *selector* connector connects either invocation connectors or other pipe or selector connectors. It makes a call into one selected connector of the connectors it connects. It can appear at any level greater than 1 in the connector hierarchy.

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):

```
public virtual void Execute (string method, object[] parameters);
public virtual void Execute (string[] methods, object[] parameters);
public virtual void Execute (int condition, string method, object[] parameters);
public virtual void Execute (int condition, string[] methods, object[] parameters)
```

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. This is done using the *Invoke* method of the *MethodInfo* class provided by .NET reflection, which is used to call methods on class instances:[2]

```
public override void Execute(string method, object[] parameters);
{
 MethodInfo methodInfo = myComponent.GetType().GetMethod(method);
 object anObj = methodInfo.Invoke(myComponent, parameters);
 Result = anObj as object[];
}
```

[2]We only present simplified code.

The invocation connector makes a method invocation with specified parameters on the component it hosts in its *Execute* method. It first obtains the *MethodInfo* class instance from the component's type, and then makes the invocation on that object returning a result.

The selector connector's *Execute* method can be passed a list of methods. Consider the case of just one method. 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 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 *parameters* will be passed to it:

```
public override void Execute(int condition, string method, object[] parameters)
{
 Connector selectedConn = myConnectors[condition];
 if(selectedConn is SelectorConnector)
 {
  object[] paramsWithoutFirstElement = GetParamsWithoutFirstElement(parameters);
  selectedConn.Execute(Convert.ToInt32(parameters[0]), method, paramsWithoutFirstElement);
 }
 else
 {
  selectedConn.Execute(method, parameters);
 }
 Result = selectedConn.Result;
}
```

If the selected connector is a selector again, the first parameter from the parameter list passed into the method gets extracted, and is used as a selection condition for that selector connector. The rest of the parameters are passed as a parameter list into the selector connector. If the selected connector itself is not a selector (but is either a pipe or an invocation connector), its *Execute* method is just called passing the method and the parameters to the connector. Finally, the invocation result is retrieved and returned. A similar *Execute* method is used for a list of methods passed to a selector connector.

The *Execute* method of a pipe connector 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. In fact, they do not have any dependencies except those shown in Figure 8, and can even be thought of as light-weight components in the system.

Since exogenous connectors form a hierarchy they 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 by the generic container introduced in the next section.

3.2 Generic Container

Having implemented exogenous connectors in a generic way, in this section we show how a generic container can be constructed that can automatically compose components into a system, using generic connectors, given a description of the control structure of the system, i.e. the connection structure for the components.

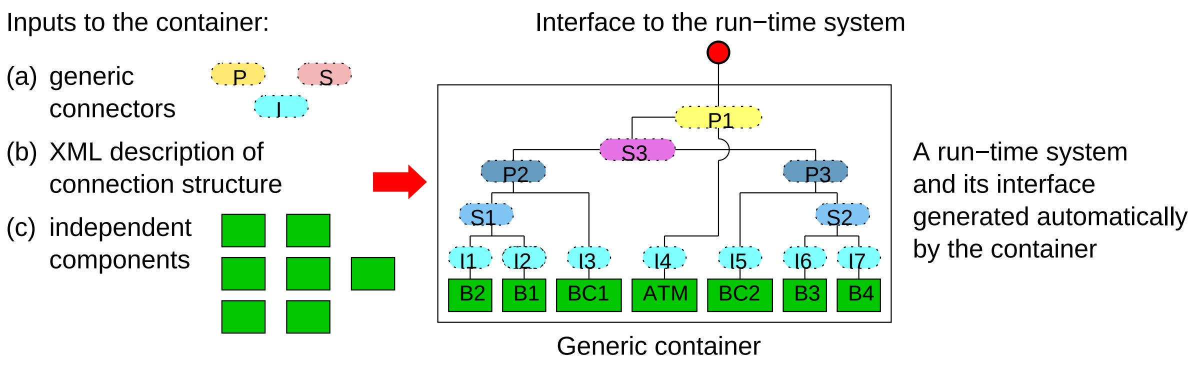

Inputs to the container:
(a) generic connectors
P
S
I
(b) XML description of connection structure
(c) independent components
Interface to the run−time system
P1
S3
P2
P3
S1
S2
I1
I2
I3
I4
I5
I6
I7
B2
B1
BC1
ATM
BC2
B3
B4
Generic container
A run−time system and its interface generated automatically by the container

Figure 9: Input to and output of the generic container.

As depicted in Figure 9 the generic container takes as input 3 kinds of entities: (a) generic exogenous connectors for component connection, (b) an XML component connection description reflecting the control structure of the system and (c) independent components not calling each other. These 3 entities are independent from each 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 description reflecting the control structure of the system in question. The output of the generic container is an ordered system constructed in accordance with the control structure description along with the top-level connector exposed to interact with the system. The system constructed provides for all control flow paths possible in the system specified by (b). A particular request to the system may not use every control flow path available. It is however ensured on the system construction that the whole control flow of the system, i.e. all possible control flow paths are available to serve all requests placed on the system through the top-level connector.

Application-*independent* templates for connectors can be created as shown in Section 3.1and reused for different applications by creating application-specific instances[3]. These 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 9. 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.

What makes the generic container possible are exogenous connectors and the strictly hierarchical nature of a system with exogenous connectors, in particular its unified system access point provided by the top-level connector. As we saw in Section 2.2, in a system with exogenous connectors there is always only one top-level connector which is responsible for initiating all control flow in the system. Therefore it is possible to construct such a system automatically (following the hierarchy) and to expose the top-level connector, which is constructed on the fly, to the outside world as an interface for interacting with the system. Thus access to the system is always normalised and non-typed.

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. Access is via the top-level component, and is therefore typed and not normalised. As far as we know, no generic container of the kind we are about to describe exists for implementing ADL systems.

The generic container we have implemented: (a) automatically creates control flow paths for a system, i.e. a component connection structure from a given system control structure description; (b) provides a runtime environment for components and connectors; and (c) takes charge of the lifecycle management of components and connectors.

[3]Template instances are not class instances in sense of object-oriented programming. When a template is instantiated it gets adapted to the current place in the connector hierarchy.

3.2.1 System Control Structure Description

In order for the generic container to be able to build up a a connector structure on the fly, it needs to process a system control structure 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 [9]; 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.

```
<?xml version="1.0" encoding="UTF-8"
standalone="yes"?> <!--W3C Schema generated by XMLSpy v2005 rel. 3 U
(http://www.altova.com)--> <xs:schema
xmlns:xs="http://www.w3.org/2001/XMLSchema"
elementFormDefault="qualified">
    <xs:element name="ExogenousADL">
        <xs:complexType>
            <xs:sequence>
                <xs:element ref="connector_types"/>
                <xs:element ref="system"/>
            </xs:sequence>
        </xs:complexType>
    </xs:element>
    <xs:element name="component">
        <xs:complexType>
            <xs:attribute name="name" use="required"/>
            <xs:attribute name="type" use="required"/>
        </xs:complexType>
    </xs:element>
    <xs:element name="connector_types">
        <xs:complexType>
            <xs:sequence>
                <xs:element name="pipe">
                    <xs:complexType>
<xs:attribute name="type" use="required"/>
                    </xs:complexType>
                </xs:element>
                <xs:element name="selector">
                    <xs:complexType>
<xs:attribute name="type" use="required"/>
                    </xs:complexType>
                </xs:element>
                <xs:element name="invocation">
                    <xs:complexType>
```

```
<xs:attribute name="type" use="required"/>
                    </xs:complexType>
                </xs:element>
            </xs:sequence>
        </xs:complexType>
    </xs:element>
    <xs:element name="invocation">
        <xs:complexType>
            <xs:sequence>
                <xs:element ref="component"/>
            </xs:sequence>
        </xs:complexType>
    </xs:element>
    <xs:element name="pipe">
        <xs:complexType>
            <xs:sequence>
<xs:element ref="pipe" minOccurs="0" maxOccurs="unbounded"/>
<xs:element ref="invocation" minOccurs="0" maxOccurs="unbounded"/>
<xs:element ref="selector" minOccurs="0" maxOccurs="unbounded"/>
            </xs:sequence>
            <xs:attribute name="name"/>
        </xs:complexType>
    </xs:element>
    <xs:element name="selector">
        <xs:complexType>
            <xs:sequence>
<xs:element ref="pipe" minOccurs="0" maxOccurs="unbounded"/>
<xs:element ref="invocation" minOccurs="0" maxOccurs="unbounded"/>
<xs:element ref="selector" minOccurs="0" maxOccurs="unbounded"/>
            </xs:sequence>
            <xs:attribute name="name"/>
        </xs:complexType>
    </xs:element>
    <xs:element name="system">
        <xs:complexType>
            <xs:sequence>
<xs:element ref="pipe" minOccurs="0" maxOccurs="unbounded"/>
<xs:element ref="invocation" minOccurs="0" maxOccurs="unbounded"/>
<xs:element ref="selector" minOccurs="0" maxOccurs="unbounded"/>
            </xs:sequence>
        </xs:complexType>
    </xs:element>
</xs:schema>
```

The XML schema we use for system control structure description is depicted in Figure 10. 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.

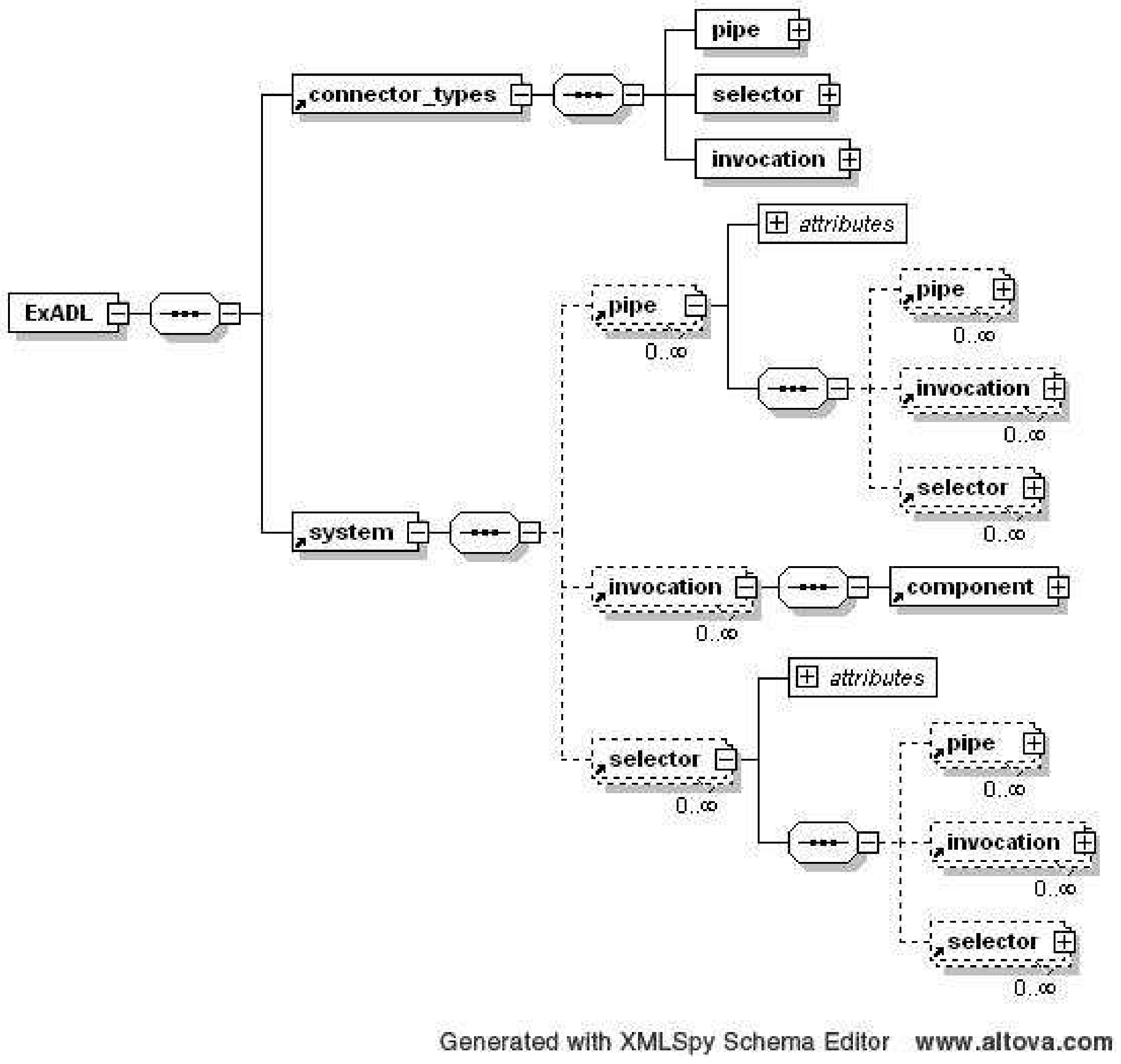


Figure 10: XML schema for system control structure description.

Connector types currently include invocation, pipe and selector connectors. They tell the container where to find and how to instantiate them.

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 (Figure 8). Note that connector types presented in Figure 8 are not the only once possible. We show only these connector types here since they are used in the Bank Example. In general, any exogenous connector types are conceivable. E.g. 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 that all those connectors can be described using system control structure description and instantiated by the generic container for exogenous connectors. I.e. the infrastructure for building systems using exogenous connectors is defined by the extensible XML schema for system control structure description and the generic container able to process it and construct system's control structure.

As an example of system control structure description, the bank system can be described by the outline in Figure 11. This can be read as: 'A pipe $P1$ contains an invocation connector and

```
<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,
          Components"/>
        </invocation>
        <selector name="S1">
          <invocation>
            <component name="B1"
            type="Components.Bank,
            Components"/>
          </invocation>
          <invocation>
            <component name="B2"
            type="Components.Bank,
            Components"/>
          </invocation>
        </selector>
      </pipe>
      <pipe name="P3">
       <invocation>
        <component name="BC2"
         type="Components.BankConsortium,
         Components"/>
       </invocation>
       <selector name="S2">
        . . .
       </selector>
      </pipe>
    </selector>
  </pipe>
</system>
```

Figure 11: System control structure description for the bank example.

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'.

3.2.2 Container Implementation

Basically, the container provides an interface *IContainer* for interaction with the outside world, which consists of: (i) a property *SystemDescription* which is used to feed the system control structure description into the container; (ii) a method *Init*() which initiates the construction of the connector hierarchy; (iii) a property *TopLevelConnector* for retrieving the top-level connector of the system; and (iv) a method *Dispose*() which disposes of the connector hierarchy.

When the container is created, it is empty and does not contain any connectors or components. The system control flow is constructed when the property *SystemDescription* is set and the method *Init* is called. During the processing of the ⟨system⟩ element, the container first retrieves the connector types and stores them for future use. A connector type is instantiated each time a specific connector occurs in the system control structure description. For example, each time a pipe element occurs in the XML description of the system, the container will create an instance of a pipe from the information stored during the processing of connector_types.

The implementation we present here is for processing pipe, selector and invocation connectors only. In reality the implementation can process any connector introduced in the connector_types section of system control structure description. In other words, hard-coded strings "pipe", "selector" and "invocation" do not occur in the real implementation, which is generic and not restricted to these connectors. This makes the container extensible to new connector types.

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 description gets validated against the XML schema and gets loaded unless the description violates the schema.

```
XmlTextReader anXmlTxtReader = new
XmlTextReader(mySystemDescription);
XmlValidatingReader
anXmlValReader = new XmlValidatingReader(anXmlTxtReader);
anXmlValReader.ValidationType = ValidationType.Schema;
XmlDocument
doc = new XmlDocument(); doc.Load(anXmlValReader);
```

Second, the ⟨connector_types⟩ element is processed, and information about the location of each connector class is stored for creating connector instances in future.

```
myPipeLoadConfig = doc.SelectSingleNode("//connector_types/pipe").
Attributes["type"].InnerText;
mySelectorLoadConfig =
doc.SelectSingleNode("//connector_types/selector").Attributes["type"].InnerText;
myInvocationLoadConfig =
doc.SelectSingleNode("//connector_types/invocation").
Attributes["type"].InnerText;
```

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,

```
switch(mySystemLoadConfig.Name)
{
 case "pipe":
  mySystem = LoadObject(myPipeLoadConfig);
  break;
 case "selector":
  mySystem = LoadObject(mySelectorLoadConfig);
  break;
 case "invocation":
  mySystem = LoadObject(myInvocationLoadConfig);
  break;
}
```

and after the whole system has been constructed, interaction with the system is possible by retrieving the property *TopLevelConnector*.

```
public Connector TopLevelConnector
{
 get
 {
  return mySystem as Connector;
 }
}
```

The complete system is created in the container beneath the top-level connector, using recursion:

```
private void LoadSystem(XmlNode theXmlNode, Connector theCurrentConnector)
{
 foreach(XmlNode anXmlNode in theXmlNode.ChildNodes)
 {
  switch(anXmlNode.Name)
  {
   case "pipe":
    PipeConnector aPipe = LoadObject(myPipeLoadConfig);
    theCurrentConnector.AddConnector(aPipe); LoadSystem(anXmlNode, aPipe);
    break;
   case "selector":
    SelectorConnector aSelector = LoadObject(mySelectorLoadConfig);
    theCurrentConnector.AddConnector(aSelector);
    LoadSystem(anXmlNode, aSelector);
    break;
   case "invocation":
    InvocationConnector anInv = LoadObject(myInvocationLoadConfig);
    theCurrentConnector.AddConnector(anInv);
    string aCompConf = anXmlNode.FirstChild.Attributes["type"].InnerText;
    anInv.Component = LoadObject(aCompConf);
    break;
    }
  }
}
```

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.

Thus the system is constructed from top to bottom and from left to right. That is, we process the nodes from the first child of the top-level connector to its last child (from top to bottom in the XML description) and, when engaging in recursion, we process the nested nodes (from left to right) until an invocation connector is found. Then the recursive call stack shrinks again, and can extend again on processing the next node.

So, the first time the recursion is entered the first child of the ⟨system⟩ node is passed into it as a first parameter, and the freshly created and empty top-level connector as a second parameter.

To illustrate how this works, consider processing the system control structure description for the bank example (Figure 11). The top-level connector is a pipe $P1$. When entered, the method *LoadSystem* will first identify that $P1$ has 2 child nodes,

```
foreach(XmlNode anXmlNode in theXmlNode.ChildNodes)
```

and will start iterating through them. It will then try to detect the connector type of the first child node,

```
switch(anXmlNode.Name)
```

and will detect that it is an invocation connector. Knowing this, it will carry out the following command sequence:

```
case "invocation":
 InvocationConnector anInv = LoadObject(myInvocationLoadConfig);
 theCurrentConnector.AddConnector(anInv);
 string aCompConf = anXmlNode.FirstChild.Attributes["type"].InnerText;
 anInv.Component = LoadObject(aCompConf);
 break;
```

An instance of the invocation connector is created from the information stored before.

```
InvocationConnector anInv = LoadObject(myInvocationLoadConfig);
```

The current connector, which is the top-level connector $P1$ in this iteration, gets assigned the newly created invocation connector.

```
theCurrentConnector.AddConnector(anInv);
```

The description of the component inside the invocation connector gets extracted and stored, and a component *ATM* gets created and assigned to the invocation connector.

```
string aCompConf =
anXmlNode.FirstChild.Attributes["type"].InnerText;
```

The component gets created and assigned to the invocation connector:

```
anInv.Component = LoadObject(aCompConf);
```

The second loop iteration (no recursion yet) will result in finding out that the connector to be created as a second child of the top-level connector $P1$ is a selector $S3$. Therefore the following code in the method LoadSystem executes:

```
case "selector":
 SelectorConnector aSelector = LoadObject(mySelectorLoadConfig);
 theCurrentConnector.AddConnector(aSelector);
 LoadSystem(anXmlNode, aSelector);
 break;
```

a selector connector gets created and stored.

```
SelectorConnector aSelector = LoadObject(mySelectorLoadConfig);
```

The top-level connector $P1$ gets assigned the selector connector as a second child.

```
theCurrentConnector.AddConnector(aSelector);
```

The method *LoadSystem* gets called recursively,

```
LoadSystem(anXmlNode, aSelector);
```

and passed the XML description of the selector connector $S3$ as well as its instance.

Now the method *LoadSystem* is entered again. First of all, a loop over $S3$'s child nodes is started. It has 2 child nodes; therefore the loop will have 2 steps. In the first step, an XML node is processed, which turns out to represent the pipe connector $P2$. So, the following code executes:

```
case "pipe":
 PipeConnector aPipe = LoadObject(myPipeLoadConfig);
 theCurrentConnector.AddConnector(aPipe);
 LoadSystem(anXmlNode, aPipe);
 break;
```

a pipe connector gets created,

```
PipeConnector aPipe = LoadObject(myPipeLoadConfig);
```

and then the recursion is initiated again by calling *LoadSystem* and passing the XML node containing the pipe and its instance.

Next an invocation connector will be created and assigned to $P2$ as a first child. The invocation connector will get the component $BC1$ injected. Afterwards, the selector connector $S1$ will get created and assigned to the selector connector $S1$ as a second child. After that the invocation connector for component $B1$ will be created and assigned to $S1$ as a first child, and the invocation connector for component $B2$ will be created and assigned to $S1$ as a second child. Then the recursive stack will diminish, and the loop started with creating the pipe $P2$ in the first iteration will be continued with the creation of the pipe $P3$ in the second and last iteration. The loop for $P3$ is similar to that for $P2$.

Now the system control flow construction is complete, and the system is ready for use via its top-level connector. In a concrete system created via the top-level connector, for any connector, an instance of the generic connector is first created and later populated with whatever connectors are beneath the connector in the system description, as we have seen before during system construction. That is why the connector is generic: it can take any number of suitable polymorphic connectors and can deal with them. Each connector has its own semantics, which is considered when writing system control structure description. The container's task is to put the connectors together according to the system control structure description. All the connectors build a type hierarchy, which makes it possible to treat them in a generic fashion by the container.

User services in a concrete system can be implemented by retrieving the *TopLevelConnector* property. When the system is used to perform such services, data and control flow starts going through the system. In this context, it is important to understand how the control flow that has been created automatically by the container works.

For instance, if a pipe (e.g. $P1$) finds an invocation connector and a selector inside itself at runtime while processing a user request, it orders the invocation connector to call a method from the method list provided as a parameter in the pipe's *Execute* method, and propagates the result to the selector connector along with the rest of the method list. The selector connector chooses a connector using the result obtained as a selection criterion, and invokes the *Execute* method of the selected connector with the method list obtained from the pipe and so on.

It should be obvious that the construction of a system's control flow can be automated by using such a generic algorithm.

The generic container lays down all possible control flow paths in the system during control flow construction, 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.

3.2.3 Usage of Container

With the generic container, the process of system construction with exogenous connectors can be mostly automated. The only code which has to be supplied by the system developer is that of the components. No other code is required to construct the system because it is constructed by the container in a generic fashion from a system control structure description on each system start. After the construction of the connector structure, the container's *TopLevelConnector* property can be used to retrieve the top-level connector created by the container. This connector allows interacting with the system.

Using the container thus fundamentally changes the programming methodology. The following steps are necessary to build a system: (i) provide independent components not calling each other, (ii) provide an XML description of system's control structure, (iii) feed the description into the container. Note that current approaches to CBSD embed the control structure into components making reuse less possible. Our components can be connected using various control structures thus enhancing component reuse.

Furthermore, the system developer has far less code to maintain because the code for connectors as well as *glue code* for connection of components to them is no longer his responsibility. Constructing n systems with an ADL compiler (ArchJava) results in n code bunches, one per system. The code for each system has to be maintained. With the generic container, there is no code bunch to be maintained since the system will be constructed by the generic container from the control structure description on every system start. The system description is used only once by ADLs, whose components like ours are accessible through method calls, to get system-specific templates for glue code, i.e. connectors, whereas our generic container uses it each time the system starts but the glue code will be constructed on the fly by the generic container. Thus, the reuse potential of our generic exogenous connectors is n times higher than that of current ADL connectors, which connect to component interfaces (ACME/ArchJava) and not event sinks (C2).

3.3 The Bank Example

Now we illustrate the use of exogenous connectors and the generic container for automated composition, using the bank example in Section 2.2.1, with the architecture described in Figure 7 (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 schema in Figure 10. We have already done this in Figure 11.

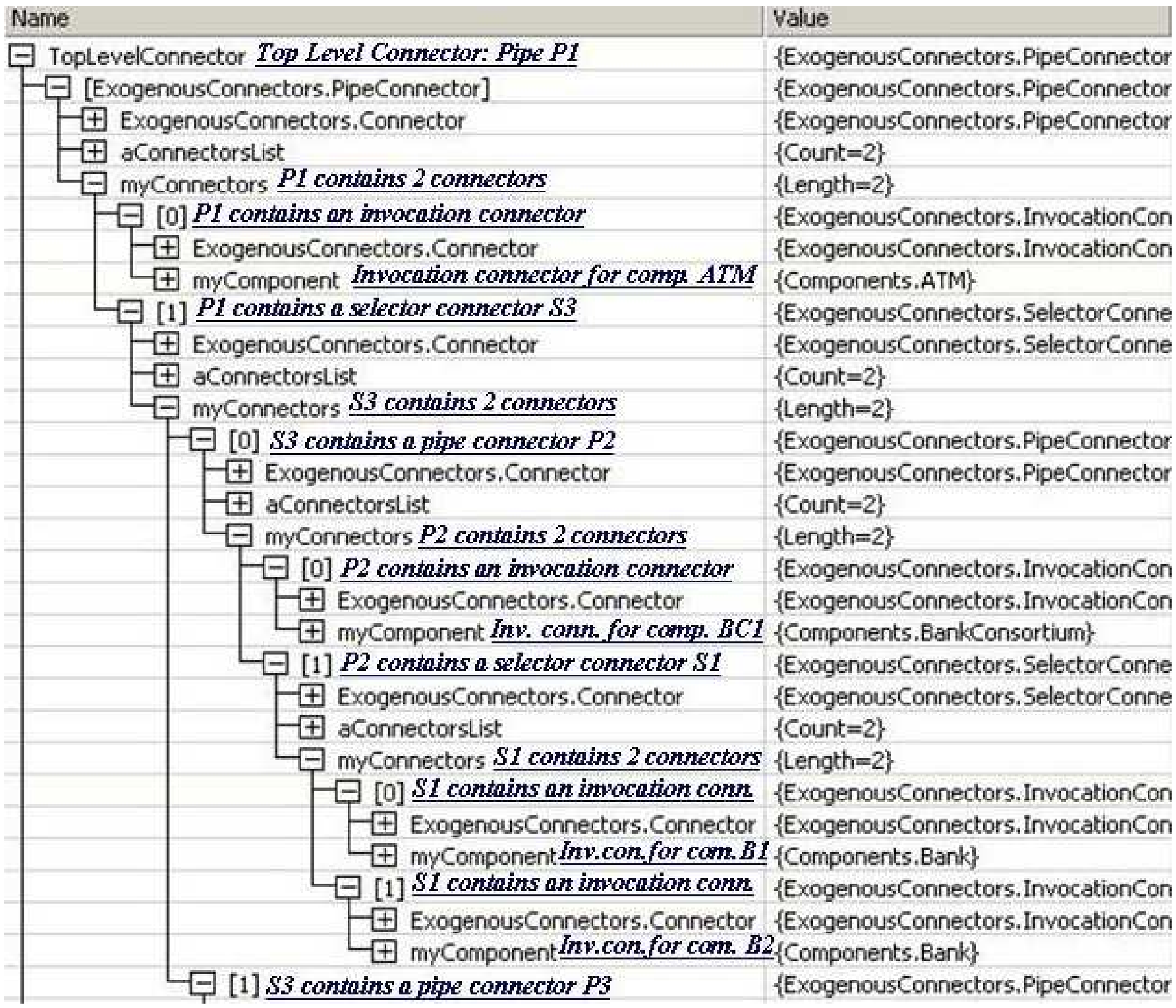
The third step is to feed this system description to the container by setting the property *SystemDescription* (and calling the *Init*() method). The result is the running system illustrated by the debugger output in Figure 12, where the connector structure for the bank system has been constructed. The generic container's *TopLevelConnector* property is explored here in the debugger. This property is exposed by the container and can be used outside, but the whole content of the container has been generated automatically.

In Figure 12, the exogenous connector instance hierarchy is shown on the left, the corresponding types on the right. The labels in italic have been added by hand to make clear what is going on at each level of the hierarchy.

The top-level connector is the pipe $P1$ here. It contains 2 connectors: the first one is an invocation connector containing the component ATM, and the second one is a selector $S3$. Since the invocation connectors with their components are always at the bottom of the hierarchy, the invocation connector does not contain any other connector. This is of course enforced not by the generic container itself but by the XML schema the system description is checked against. The selector $S3$, in contrast, contains 2 connectors. The first one is a pipe connector $P2$.

$P2$ contains an invocation connector for component $BC1$ as well as a selector connector $S1$. The selector $S1$ in turn contains 2 connectors, which are both invocation connectors for components $B1$ and $B2$ respectively.

Clearly the running bank system is precisely the system specified by the system description



```
Name                                                                  Value
[-] TopLevelConnector Top Level Connector: Pipe P1                    {ExogenousConnectors.PipeConnector
  [-] [ExogenousConnectors.PipeConnector]                             {ExogenousConnectors.PipeConnector
    [+] ExogenousConnectors.Connector                                 {ExogenousConnectors.PipeConnector
    [+] aConnectorsList                                               {Count=2}
    [-] myConnectors P1 contains 2 connectors                         {Length=2}
      [-] [0] P1 contains an invocation connector                     {ExogenousConnectors.InvocationCon
        [+] ExogenousConnectors.Connector                             {ExogenousConnectors.InvocationCon
        [+] myComponent Invocation connector for comp. ATM            {Components.ATM}
      [-] [1] P1 contains a selector connector S3                     {ExogenousConnectors.SelectorConne
        [+] ExogenousConnectors.Connector                             {ExogenousConnectors.SelectorConne
        [+] aConnectorsList                                           {Count=2}
        [-] myConnectors S3 contains 2 connectors                     {Length=2}
          [-] [0] S3 contains a pipe connector P2                     {ExogenousConnectors.PipeConnector
            [+] ExogenousConnectors.Connector                         {ExogenousConnectors.PipeConnector
            [+] aConnectorsList                                       {Count=2}
            [-] myConnectors P2 contains 2 connectors                 {Length=2}
              [-] [0] P2 contains an invocation connector             {ExogenousConnectors.InvocationCon
                [+] ExogenousConnectors.Connector                     {ExogenousConnectors.InvocationCon
                [+] myComponent Inv. conn. for comp. BC1              {Components.BankConsortium}
              [-] [1] P2 contains a selector connector S1             {ExogenousConnectors.SelectorConne
                [+] ExogenousConnectors.Connector                     {ExogenousConnectors.SelectorConne
                [+] aConnectorsList                                   {Count=2}
                [-] myConnectors S1 contains 2 connectors             {Length=2}
                  [-] [0] S1 contains an invocation conn.             {ExogenousConnectors.InvocationCon
                    [+] ExogenousConnectors.Connector                 {ExogenousConnectors.InvocationCon
                    [+] myComponent Inv.con.for com.B1                {Components.Bank}
                  [-] [1] S1 contains an invocation conn.             {ExogenousConnectors.InvocationCon
                    [+] ExogenousConnectors.Connector                 {ExogenousConnectors.InvocationCon
                    [+] myComponent Inv.con.for com. B2               {Components.Bank}
          [-] [1] S3 contains a pipe connector P3                     {ExogenousConnectors.PipeConnector
```

Figure 12: Running the bank system.

that is fed into the container. Of course a nice user interface could present the output in a better way, but the whole system is constructed automatically by the container.

Now we briefly explain how this 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 retrieving the *TopLevelConnector* of the bank system, as follows:

```
myBankExample.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 passes (at which connector and component) was specified before in the system description.

For the get balance operation, the control flow involved is shown in Figure 13. So the

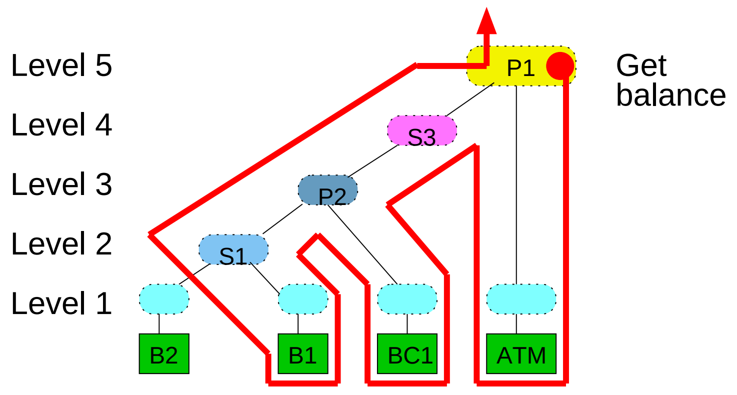


Figure 13: Control flow for get balance.

sequence of events can be outlined as follows:

1. The pipe $P1$ calls ATM's invocation connector to look up a bank consortium number for the card inserted. ATM looks up the bank consortium number in a data base and returns it to its invocation connector, which returns the number to $P1$. Then $P1$ takes the value supplied by the ATM's invocation connector and passes it on to the selector $S3$.

2. $S3$ selects one of the connectors it contains and makes an invocation in the selected connector. It looks at the value supplied by $P1$, which is a bank consortium number, and chooses the pipe $P2$ to be called.

3. $P2$ has 2 connectors in it. The first one is $BC1$'s invocation connector, so $P2$ invokes it to request the bank number to be used. $BC1$ retrieves the bank number from the data base and returns it to its invocation connector, which in turn returns it to $P2$.

4. $P2$ passes the bank number on to $S1$, which chooses $B1$'s invocation connector to be called, to get the balance requested by the user. $B1$ retrieves the balance from the data base and returns the value to its invocation connector, which returns it to the selector $S1$.

5. $S1$ returns the balance to $P2$, which returns it to $S3$, which returns it to the top-level connector $P1$.

Note that the control flow for get balance operation does not use all possible control flow paths laid down by the container on system construction but rather uses a part of them. Figure 9 shows that the container constructs all the possible control flow paths in the system. Figure 13 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.

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:

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

Withdraw $100 from account the card 4711 belongs to:

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

Finally, the bank system can be easily changed without writing any code. For instance, suppose we want to introduce an additional bank consortium, $BC3$, and to allocate 2 new bank branches, $B5$ and $B6$, to it. This can be done by simply adding the following XML snippet to the system description:

```
<pipe name="P3">
  <invocation>
    <component name="BC3"
    type="Components.BankConsortium, Components"/>
  </invocation>
  <selector name="S2">
    <invocation>
      <component name="B5"
      type="Components.Bank, Components" />
    </invocation>
    <invocation>
      <component name="B6"
      type="Components.Bank, Components" />
    </invocation>
  </selector>
</pipe>
```

Of course any data base the components operate on will have to be extended as well, but the system developer does not have to write any additional code to extend the system itself.

Conversely, to shrink a system, it only requires an XML snippet omitted in the specification.

4 Evaluation

The main contribution of the container is component composition by automated system control flow construction. Whereas other approaches to component-based system composition involve new code introduced by the connectors, our approach not only automates connector construction but also eliminates the necessity of writing and maintaining glue code. So, all the control flow paths in a system with exogenous connectors are automatically generated by the container on the fly.

Beyond simplified system construction and deployment there are several other advantages of employing the generic container for exogenous connectors:

- Exogenous connector hierarchy makes the system very modular because at any level there is a top-level connector for that level. Therefore we can treat any level of the hierarchy independently. This allows for simplified system testability as it is possible to test every connector with all its child connectors independently. With the generic container this becomes even easier because it only takes a corresponding XML snippet to be fed into the generic container to test a system part.

- The amount of code in the system is reduced to the necessary minimum as glue code for component connection is managed by the container through customised instances of generic connectors. The system assembler does not have to write and maintain any glue code any more, which simplifies system maintenance.
- The generic container allows for easy system extension by adding new XML snippets to the system control structure description.
- System adaptation is also simplified. For example, components can be easily exchanged by changing the system description. Changes take effect on the next system start.
- Overall, the system is robust to extensions and change because no code changes at all can be required to modify the system. Some changes require coding, of course.
- Certain system semantics can be enforced and checked before the system construction by checking XML system control structure description against XML schema.
- As the container processes XML Files that conform to the XML schema, it can be implemented in nearly any language. The language needs to support polymorphism, late-binding and loading as well as introspection and reflection mechanisms, which are the key features deployed by the component-oriented programming methodology identified by [28] on Page 457.
- Exogenous connectors construct a hierarchy of connectors. The top level connector is the origin of the hierarchy and initiates all the control in the system. Essentially, there is no other access to the system than via the top level connector. The top level connector has an interface, which allows the system to be accessed via a web service. It has always been a problem to access an object-oriented interface which can contain not only simple types but rather self-defined types via a web service because of marshaling the values of self-defined types. Simple types can be transmitted via network using existing frameworks but self-defined types have to marshaled by the system developer itself, which can be a non-trivial task. Since in a system with exogenous connectors all components are at the bottom of the hierarchy and the top level connector is the only entity exposed to the outside, it is easier to access such a system via web service self-defined types in component interfaces notwithstanding.

As far as we know, the container we have implemented is unique because it generates control flow of systems consisting of independent components automatically. ADLs do not have generic and hierarchical connectors. Even XML-based ADLs like xADL 1.1 [16] and xADL 2.0 [8] do not have a generic container like ours. In component models that do have containers, viz. JavaBeans, EJB and CCM [19], the containers only instantiate and host components, and do not generate systems control flow paths automatically at runtime. Work on runtime reconfiguration, e.g. [20], is usually based on such containers, and does not support automated component composition in the way exogenous connectors do.

The following containers exist in today's component models and ADLs: EJB container for hosting EJBs, CCM container for hosting CCM components, Bean Box for hosting Java Beans

component model	access to component	component composition by container	control origin	separation of business logic from user interface
EJB	by method call	no	component	yes
CCM	by method call	no	component	yes
Java Beans	by event	yes	component	no
C2	by event	yes	component	no
Exogenous	by method call	yes	connector	yes

Table 1: Comparison of current containers with the container for exogenous connectors.

and Bootstrapper in C2 ADL. All these containers can instantiate and manage components they host. In the following we compare all these containers with our container for exogenous connectors.

EJB container. An EJB container instantiates enterprise beans and provides services to components like persistence management or look up for other beans via JNDI etc. The components are composed (get to know each other) at their design time through method calls and the container does not do any composition.

CCM container A CCM container is like EJB container. It instantiates CORBA components and provides them with various services. It is also not responsible for component composition. Component composition is done using method calls at component design time.

Java Bean Box. Java Bean Box loads independent Java Beans and composes them by compilation of corresponding adaptor classes. A Java Bean emits events, which are caught by another bean if the corresponding adaptor class is available. The communication in Java Beans is always asynchronous using events. Composition is done at bean deployment time using system-specific adaptor classes, which is in contrast to our generic connectors.

C2 Bootstrapper. The Bootstrapper in C2 ADL is responsible for instantiation of components and bus connectors and their composition. The bus connector in C2 is the only one available and is generic. It is instantiated whenever a connection between C2 components is needed. C2 components originate events to each other, which are dispatched to component event sinks. The component communication is like in Java Beans asynchronous through event dispatching.

Table 1 summarises the component models with containers and shows the differences to our proposed model. The comparison shows clearly that our system is the only one where not components but connectors originate control in the system. Furthermore, our system is the only one where the container does component component composition of components which are accessible via method calls. Systems whose components are accessible through method calls (EJB and CCM) do not use container for component composition. Their components are tightly coupled as they use method calls from within components to connect them. Our system is distinct as although our components are accessible through methods calls, they are nevertheless independent of each other through the use of exogenous connectors. Furthermore, our container does component composition by generating control flow of the system automatically (Figure 15).

Moreover, components in systems, which have containers for component composition (Java Beans and C2), encapsulate business logic with UI, making it impossible to represent the business logic in a different way as suggested by the classic Model View Controller pattern [11]. Our components do not have to contain UI logic as it can be constructed above the top-level connector. This enables system developer to present the business logic in several ways using different user interfaces. That is, the reuse potential of our components is higher than those in Java Beans and C2.

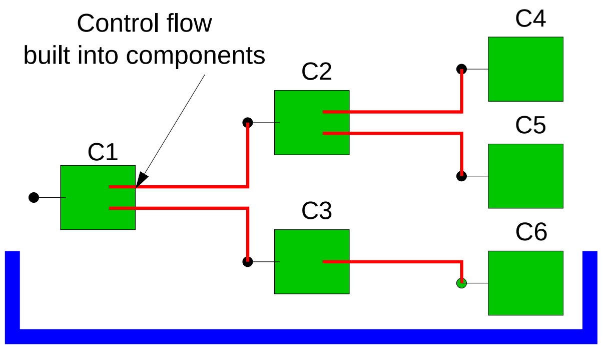


Figure 14: Component instantiations in current containers.

To summarise, containers like EJB and CCM can be illustrated as shown in Figure 14. They construct hierarchies of dependent components. The links are set up by components even before component deployment into containers. We take another approach and let our container construct hierarchies of connectors, thus leaving the components independent. As components do not know each other at all, the links in Figure 15 have to be set by the container.

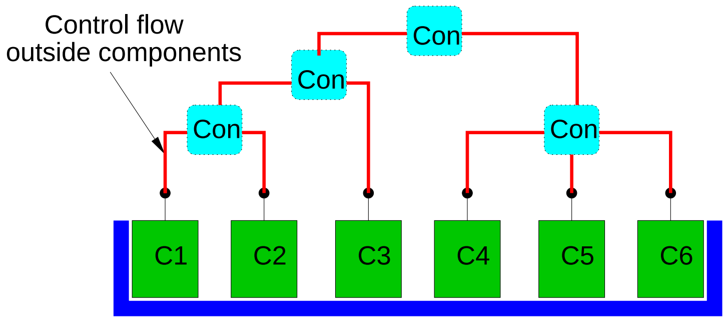


Figure 15: Container for exogenous connectors.

Our container is generic, which means that we can be confident about its wide applicability, even though we have only demonstrated its promise with a single example. In this regard, the choice of .NET as an implementation platform is a particularly good one. It allows components implemented in different .NET languages to be loaded into our container, thus enabling programming language interoperability. For instance, in the bank example, a bank component can be implemented in VB.NET, a bank consortium in Managed C++, an ATM in Eiffel, and the whole lot can be loaded into the generic container programmed in C#. Furthermore, our container could also run in Linux, using the CLR provided by Mono [22].

It could be argued that it does not really matter to the system developer how many connectors the system consists of, since the generic container automatically takes care of them. However, the foot print of the system gets larger, the more connectors there are in the system. Using the generic container as a loader of the system can increase system start time because connectors as well as components are loaded on the fly. However, if connectors and components are not scattered over many binaries, this time can be significantly reduced because a binary with its entire content is loaded into the container only once.

Another issue is whether exogenous connection can be used to describe layered systems. Exogenous connectors as they are presented here somewhat prevent software layering. As shown above, all the method names to be called on the components at the lowest level of the hierarchy have to be supplied as parameters of the top level connector's Execute method. For example, deposit \$100 onto account the card 4711 belongs to:

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

In a truly layered system a layer doesn't know anything about layers which are not immediately next to it. So, software layer N only knows software layer $N-1$ and knows nothing about the layers $0 < K < N-1$. In the system with exogenous connectors there are no direct dependencies of the highest layer on the lowest layer containing components in terms of calling their methods but there are dependencies on method names of component's at the lowest layer to be called by invocation connectors. Moreover, the more components are involved the more method names have to be supplied to the top level connector as strings, thus increasing coupling. Interestingly, the number of layers does not contribute to coupling strength.

ADLs (in particular C2) can describe layered systems well, as illustrated by Figure 5 (a) and (b). It will be interesting to study and compare the dependencies between ADL layers and those between levels in the exogenous connector hierarchy.

Other issues include concurrency and distribution. Currently we have only synchronous calls, and we have not investigated distributed systems. It will be interesting to see whether exogenous connectors can be used for asynchronous calls as well, and if so, whether it is (also) possible to automate asynchronous composition. In this context, the use of inheritance for the connector hierarchy and reflection for method calls may be problematic. Higher-order languages for concurrency and (distributed) composition may be more appropriate.

Finally, we should also examine how our approach relates to work in web services and service-oriented architectures for web information systems, where service orchestration seemingly exhibits a flavour of exogenous connection.

5 Conclusion

In [17] we introduce exogenous connectors for encapsulating control, believing that by separating control from computation, we can make compositional reasoning more tractable. However, if exogenous connectors are not practical at all, then we will not even get a chance to verify our claim. So in this report we have set out to show that not only are exogenous connectors practical, but they in fact allow for automated component composition. The generic container we have described in this report provides the evidence.

Of course, this is only a start, and in the long run, we do want to be able to reason about all aspects of systems with exogenous connectors. Clearly, the container we have implemented will have a crucial role to play in reasoning. Therefore, in the mean time, we will continue to work on (exogenous connectors and) the generic container, and investigate what aspects of reasoning the container can support.

One thing we are working on are so-called deployment contracts for software components. Currently, our components are just classes, so they cannot be properly unloaded on demand. A deployment contract can provide various kinds of deployment information, e.g. dependencies, security infrastructure, invocation types, licensing mode, etc. With this kind of information, we will be able to reason about component composition by providing a reasoning framework for static conflict detection and prevention. This framework can be employed during the creation of composite components as well as during component selection for the system. It can also be applied while loading components into the system, which is a special case deserving attention: using the framework will increase system start time, but it could be crucial for highly secure systems.

Harnessing the components in a container makes it possible to investigate versioning issues. Component versioning information can be included in system descriptions and used for version control. Moreover, connectors themselves could also be versioned. A combinatorial versatility needs to be investigated, but having the generic container, which controls and harnesses the whole system, is a step in the right direction because it is the very part of the system which can enforce some rules.

The container could also propagate system-wide events to components, An example of such an event is dynamic system reconfiguration (e.g. [20]). However, such event handling tightens the coupling between the components and the container, which rather defeats the advantage of independent components that exogenous connectors proffer.

References

[1] J. Aldrich, C. Chambers, and D. Notkin. Architectural reasoning in ArchJava. In *Proc. 16th ECOOP*, pages 334–367. Springer-Verlag, 2002.

[2] J. Aldrich, C. Chambers, and D. Notkin. ArchJava: Connecting software architecture to implimentation. In *Proc. ICSE 2002*, pages 187–197. IEEE, 2002.

[3] F. Arbab. The IWIM model for coordination of concurrent activities. In *LNCS 1061*, pages 34–56. Springer-Verlag, 1996.

[4] ArchJava web page. `http://archjava.fluid.cs.cmu.edu/index.html`.

[5] 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.

[6] D. Box. *Essential COM.* Addison-Wesley, 1998.

[7] E. Bruneton, T. Coupaye, and J. Stefani. The Fractal component model. Technical Report Draft-Version 2.0-3, The ObjectWeb Consortium, 2004.

[8] E. Dashofy, A. van der Hoek, and R. Taylor. A highly-extensible, XML-based architecture description language. In *Proc. WICSA*, pages 103–112. IEEE Computer Society, 2001.

[9] L. Dykes, E. Tittel, and C. Valentine. *XML Schemas.* Sybex Inc, 2002.

[10] R. Englander. *Developing Java Beans.* O'Reilly & Associates, 1997.

[11] E. Gamma, R. Helm, R. Johnson, and J. Vlissades. *Design Patterns – Elements of Reusable Object-Oriented Design.* Addison-Wesley, 1994.

[12] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. The courier pattern. *Dr. Dobb's Journal*, Feburary 1996.

[13] D. Garlan, R. Monroe, and D. Wile. Acme: Architectural description of component-based systems. In G. Leavens and M. Sitaraman, editors, *Foundations of Component-Based Systems*, pages 47–68. Cambridge University Press, 2000.

[14] R. Haefel. *Enterprise JavaBeans.* O'Reilly & Associates, 3rd edition, 2001.

[15] J. Ivers, N. Sinha, and K. Wallnau. A Basis for Composition Language CL. Technical Report CMU/SEI-2002-TN-026, CMU SEI, 2002.

[16] R. Khare, M. Guntersdorfer, P. Oreizy, N. Medvidovic, and R. N. Taylor. xADL: Enabling architecture-centric tool integration with XML. In *Proc. 34th Hawaii Int. Conf. on System Sciences*, 2001.

[17] K.-K. Lau, P. Velasco Elizondo, and Z. Wang. Exogenous connectors for software components. In *Proc. 8th CBSE, LNCS 3489*, pages 90–106, 2005.

[18] K.-K. Lau and Z. Wang. A survey of software component models. Pre-print CSPP-30, School of Computer Science, The University of Manchester, April 2005.

[19] K.-K. Lau and Z. Wang. A taxonomy of software component models. In *Proc. 31st Euromicro Conference.* IEEE Computer Society Press, 2005.

[20] J. Matevska-Meyer, W. Hasselbring, and R. Reussner. Software architecture description supporting component deployment and system runtime reconfiguration. In *Proc. 9th Int. Workshop on Component-oriented Programming 2004.*

[21] N. Mehta, N. Medvidovic, and S. Phadke. Towards a taxonomy of software connectors. In *Proc. 22nd ICSE*, pages 178–187. ACM Press, 2000.

[22] Mono – .NET CLR for Linux. `http://www.mono-project.com/About_Mono`.

[23] 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 Int. Conf. on Component Deployment*, pages 200–209. ACM Press, 2002.

[24] OMG. *UML 2.0 Superstructure Specification.* `http://www.omg.org/cgi-bin/doc?ptc/2003-08-02`.

[25] OMG. *CORBA Component Model, V3.0*, 2002. `http://www.omg.org/technology/documents/formal/components.htm`.

[26] F. Plasil, D. Balek, and R. Janecek. SOFA/DCUP: Architecture for component trading and dynamic updating. In *Proc. ICCDS98*, pages 43–52. IEEE Press, 1998.

[27] M. Shaw and D. Garlan. *Software Architecture: Perspectives on an Emerging Discipline.* Prentice Hall, 1996.

[28] C. Szyperski, D. Gruntz, and S. Murer. *Component Software: Beyond Object-Oriented Programming.* Addison-Wesley, second edition, 2002.

[29] R. N. Taylor, N. Medvidovic, K. M. Anderson, E. J. W. 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.

[30] 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.