Composite Connectors for Composing Software Components

Kung-Kiu Lau, Ling Ling, Vladyslav Ukis and Perla Velasco Elizondo

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

Abstract. In a component-based system, connectors are used to compose components. Connectors should have a semantics that makes them simple to construct and use. At the same time, their semantics should be rich enough to endow them with desirable properties such as genericity, compositionality and reusability. For connector construction, compositionality would be particularly useful, since it would facilitate systematic construction. In this paper we describe a hierarchical approach to connector definition and construction that allows connectors to be defined and constructed from sub-connectors. These composite connectors are indeed generic, compositional and reusable. They behave like design patterns, and provide powerful composition connectors.

1 Introduction

A component-based system can be described as a software architecture [14] with components (boxes) and connectors (lines). Components represent parts of the system, while connectors represent interactions between components. Connectors are therefore composition operators for the components.

Clearly, in a component model, the ease of building systems and reasoning about the process depends directly on the varieties of connectors available and their semantics. A crucial question therefore is how to define and construct suitable connectors.

Connectors should have a semantics that makes them simple to construct and use. At the same time, their semantics should be rich enough to endow them with desirable properties such as genericity, compositionality and reusability. For connector construction, compositionality would be particularly useful, since it would allow connectors to be constructed a systematic manner.

In this paper we describe a hierarchical approach to connector definition and construction. Using a set of basic exogenous composition connectors, we can define and construct a composite connector as a composition of the basic connectors. The resulting composite connectors are indeed generic, compositional and reusable.

Because our basic connectors define control structures, our composite connectors represent composite control structures, or composite control flow patterns. As such, they behave like certain design patterns [6], and provide powerful composition operators that can be used to perform complicated compositions involving many components all in a single step.

M. Lumpe and W. Vanderperren (Eds.): SC 2007, LNCS 4829, pp. 266–280, 2007.
© Springer-Verlag Berlin Heidelberg 2007

The paper is organised as follows. In Section 2 we describe related work on the issue of composite connectors. In Section 3 we briefly describe the concept of exogenous composition connectors and present a basic set of these connectors. In Section 4 we explain composite connectors in detail and how they are implemented. In Section 5 we show how they can be used in practice. Finally, in Section 6 we discuss our approach for creating composite connectors.

2 Related Work

As far as we know, our approach to composite connectors is new and unique. There are two main related areas: *software architectures* [14] and *coordination languages* [13].

In *software architectures*, there is work on compositional approaches to connector construction, but it does not construct connectors from sub-connectors. Rather it tries to construct only a *single* connector. This construction consists in a composition of elements with the desired properties, yielding a new connector; or a composition of the necessary adaptations or transformations of an existing connector to achieve these properties. In [15], an ADL (Architecture Description Language [11]) connector can be adpated by composing a set of transformations. The transformations can modify the connector's properties, e.g. protocol, data policy. Typically they also change the code of the components involved since connector code is embedded in component code.

In [4,5] a connector is composed from a set of *connector elements*. The elements model certain non-functional properties of some basic connector types supported in middleware technologies. In [10], an existing connector's aspects, e.g. security, monitoring, etc., can be specified separately and then composed and integrated with the connector.

These approaches only deal with the construction of a single connector. Furthermore, in these approaches either connectors are not distinct from components, e.g. [15], or when they are, their implementations are customized solutions for a specific system [4,5]. Therefore in all these approaches, both components and connectors cannot be reused.

In *coordination languages*, composite connectors can be constructed. In these languages, connectors are used to coordinate component interactions. Compared to ADL connectors, these connectors can represent much more sophisticated coordination policies for sets of components. In the coordination language Reo [2,1] connectors are composed of channels. The channels are compositional, and therefore composite connectors can be defined.

However, Reo composite connectors are very different in nature from our composite connectors. In Reo, components only perform I/O operations, and connectors are data channels. Consequently, a composite connector in Reo is not a control structure, and so it differs form our composite connector. In particular, a Reo composite connector does not behave like any design pattern.

3 Exogenous Composition Connectors

Our approach is based on exogenous connectors. In this section we briefly explain what exogenous connectors are, and how they are used as composition operators for software components.

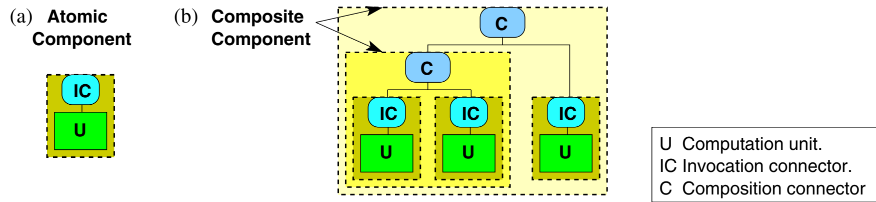

Fig. 1. Atomic and composite components

Exogenous connectors are defined within the context of our component model [8]. In our component model, there are two kinds of basic entities: (i) *exogenous connectors* [7] and (ii) *computation units*. Components are constructed from exogenous connectors and computation units. A computation unit performs only computation (by providing a set of methods) and does not invoke any computation outside itself. Exogenous connectors coordinate all the computation performed by components.

There are two kinds of components: *atomic* and *composite*. An atomic component (Fig.1 (a)), consists of a computation unit (U) and an exogenous connector for invoking the methods in the computation unit.

This connector is called an *invocation connector* (IC). A composite component is composed from (atomic or composite) components by using a *composition connector* (C in Fig. 1 (b), which shows two composite components). This is an exogenous connector that defines a piece of control that coordinates all the calls to the methods in the sub-components. In a system, the set of all the composition connectors encapsulate all the control in the system. For example, a *Sequencer* connector that composes two atomic components A_1 and A_2 can call a method m_1 in A_1, and a method m_2 in A_2, in that order. A *Pipe* connector composing A_1 and A_2 behaves similarly, but can also pass the result of m_1 to A_2 and use it in calling m_2. Components do not initiate any control, and just provide services when invoked by the connectors.

Every component thus has a top-level connector: this is either an invocation connector (for an atomic component) or a composition connector (for a composite component). This connector acts as an interface for the component, and is also used by other connectors for composition.

In [7], we have introduced these basic exogenous composition connectors which encapsulate different control structures that are necessary for building systems. The control encapsulated in these connectors corresponds to the three standard control structures: *sequencing*, *branching* and *looping*; therefore this set of connectors is Turing complete [12,3].

3.1 A Hierarchy of Composition Connectors

Exogenous composition connectors are defined in a hierarchical way (as can be seen in Fig. 1). For example, a *Sequencer* connector, or a *Pipe* connector, that composes two atomic components A_1 and A_2 is clearly defined in terms of the invocation connectors in A_1 and A_2.

In general, exogenous composition connectors form a hierarchy built on top of invocation connectors for atomic components. The lowest level (level 1) of composition

connectors connect invocation connectors, and the second-level (level 2) composition connectors are of variable arities and types. In general, composition connectors at any level can be of variable arities; composition connectors at any level higher than 1 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 $(n-1)$. In particular, the types of the former are defined in terms of the types of the latter. The connector type hierarchy can be defined in terms of dependent types and polymorphism as follows (omitting methods and their parameters):

Basic types: Atomic Component, Result;

Connector types:

$$I \equiv \text{Atomic Component} \longrightarrow \text{Result};$$

$$L1 \equiv I \times \ldots \times I \longrightarrow \text{Result};$$

$$\text{For } 1 < i \leq n, \quad Li \equiv L(j_1) \times \ldots \times L(j_m) \longrightarrow \text{Result, for some } m$$

$$\text{where } j_k \in \{1, ..., (i-1)\} \text{ for } 1 \leq k \leq m,$$

$$\text{and } L(i) = \begin{cases} L1 \ , \ i = 1 \\ L2 \ , \ i = 2 \\ \vdots \\ Ln \ , \ i = n. \end{cases}$$

where I is the Invocation Connector type, and Li is the Level-i Composition Connector type, for $1 \leq i \leq n$.

Accordingly we have implemented composition connectors as a hierarchy of classes[1] which extend a common superclass called Connector (Fig. 2).

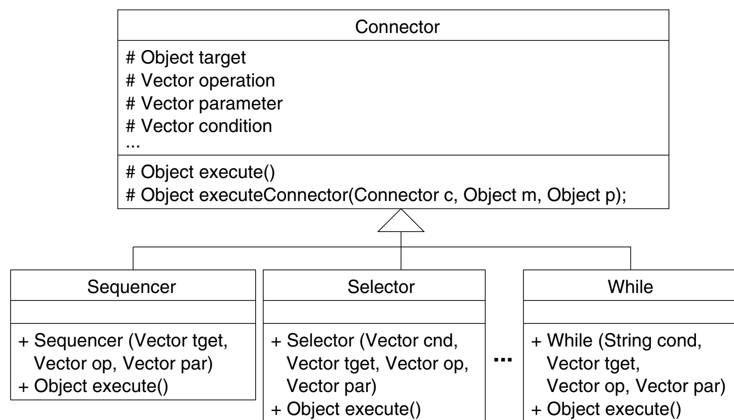

Fig. 2. Hierarchy of composition connector classes with the superclass Connector

At any level of the hierarchy, a connector can be defined in a generic manner as a class that extends and overrides selected methods of the superclass Connector. We have implemented a set of five basic composition connectors: *Sequencer* and *Pipe*, *Selector*,

[1] We have two implementations, one in Java and another in .NET C#.

and *While* and *Repeat*, which correspond to sequencing, branching and looping control respectively.

Each connector is made up of a *signature* and *code*. The signature, implemented by the connector's *constructor*, indicates how the connector can be used. The code implements the connector's functionality, and is defined as a method called execute.

As shown in Fig. 2, the constructors of all the connector receive a common set of parameters, i.e. tget, op and par. For a connector, tget specifies the set of connectors it is connected to; op is the set of operations to be executed via those connectors; and par is the set of parameters required to support the executions. The implementations of the constructors are all similar; the constructors only verify the type and number of the arguments they receive, and store them into the corresponding superclass fields, i.e. target, operation and parameter.

The `execute` method of a composition connector is inherited from the Connector superclass and overridden by the connector class. The `execute` methods of all the connectors are very similar and only differ in the specific code required for the control scheme they encapsulate, e.g. a *Selector* connector requires some code for evaluating its condition. All `execute` methods call the `executeConnector` method implemented in the Connector superclass. This method contains the code for executing any connector at any level of the hierarchy.

Fig. 3(a) shows an outline of the code for the `executeConnector` method. This illustrates the hierarchical execution of connectors. First, the subtype of the connector is identified via specific supporting functions arranged in an "if-then-else" control structure. Once the connector subtype is identified, it is stored in a variable of this subtype by casting it. For example, if the connected sub-connector is of type *Sequencer*, it needs to be cast to this type, which is a subtype of Connector as shown in Fig. 2. Finally, the connected connector is executed by calling its corresponding `execute` method. This process is repeated for all the connected connectors in a hierarchy until the invocation connectors are encountered.

(a)

```
class Connector {
 ...
 Object executeConnector(Object connToExecute,...){
   ...
   if (isInvocation(connToExecute)){
       Invocation ic = (Invocation) connToExecute;
       ...
       r = ic.execute(oper, params);
    else if (isSequencer(connToExecute)){
       Sequencer seq = (Sequencer) connToExecute;
       ...
       r = seq.execute();
    else if (...){
       ...
    } else if (isWhile(connToExecute)){
       While whi = (While) connToExecute;
       ...
       r = whi.execute();
    }
    return r;
  }
}
```

(b)

```
class Invocation extends Connector {
  private Object cu;
  ...
  public Object execute(
  Method operationToExecute, Object[] par){
    r = operationToExecute.invoke(cu, par);
    return r
  }
}
```

Fig. 3. Outline of the codes for (a) the executeConnector method in the Connector superclass and (b) the execute method in the Invocation connector class

Invocation connectors are not composition connectors, and so their `execute` methods are different. Fig. 3 (b) shows an outline of the code for the `execute` method of a invocation connector. This method requires two arguments: `operationToExecute` and `par`, which correspond to the name of the operation to invoke in the computation unit, and its parameters, respectively. We use the `invoke` method provided by the class Method in the java.lang.reflect package to dynamically execute the required operation in a computation unit (`cu`).

The hierarchical nature of composition connectors means that every system has one, and only one, top-level connector, which initiates control flow for the entire system calling the execute methods of connected connectors following a top-down approach. To illustrate this, consider the architecture with exogenous composition connectors in Fig. 4. The architecture corresponds to a Coffee Machine system. For simplicity and clarity, we have not explicitly distinguished between atomic and composite components in the architecture. The Coffee Machine consists of a hierarchical structure of composition connectors (*Sequencers SQ2* and *SQ1*, *Selector SEL* and *Pipe PIPE*) representing the system's control flow, sitting on top of independent components (Card Manager, Cash Manager, Coffee Maker, Cup Dispenser, Coffee Dispenser, Water Dispenser, Milk Dispenser and Sugar Dispenser) that provide the computation performed by the system. The execution of the system starts with the composition operator at the highest level, namely the *Sequencer SQ2*. The customers of the system can pay for a

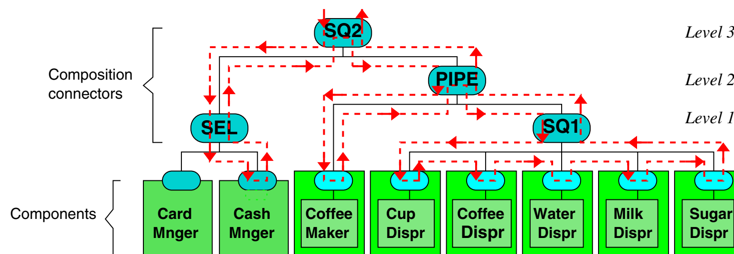

Fig. 4. An architecture with our basic composition operators

coffee either by cash or by card. Consider the use case of buying a coffee with cash. The control flow path for this is shown by the dotted line in Fig. 4. The first action is the execution of the level-3 connector *SQ2*, which firstly calls the level-1 *Selector SEL*. The latter chooses the component Cash Manager, and invokes the required method in it to process the transaction. Then, *SQ2* calls the level-2 *Pipe PIPE*, which invokes one of the operations in the component CoffeeMaker to get from a recipe the amount of each ingredient for the selected product. The amounts are passed through *PIPE* to *SQ1* which uses them as parameters for invoking methods in each one of the dispenser components. Finally the control flow goes back across the composition hierarchy until it reaches *SQ2*, whereupon the transaction is completed. If any data is generated by the dispenser, e.g. an error or success code, it is also transmitted back across the hierarchy with the control flow.

4 Composite Composition Connectors

The hierarchical nature of composition connectors means that the connectors themselves can be composed into composite composition connectors. In this section, we explain composite connectors in detail.

4.1 Composite Connectors are Patterns

It should be clear from the previous section that a set of connectors that are interconnected can be regarded as a single composite connector *CC*, which in turn can be used in hierarchical composition subsequently. It should also be clear that *CC* is a *pattern*, since it represents a composite control structure that composes a set of components.

For example, in the Coffee Machine example (Fig. 4), the level-2 *Pipe PIPE* and level-1 *Sequencer SQ1* can be composed into a composite composition connector, as shown in Fig. 5.

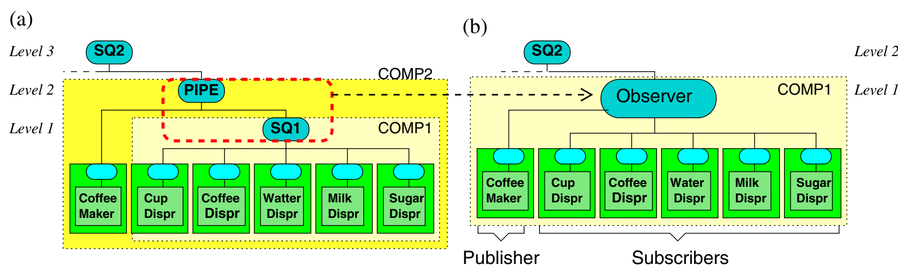

Fig. 5. Coffee Machine with (a) basic connectors and (b) composite connector Observer

This composite connector is equivalent to the object-oriented *Observer* design pattern [6]. This is because it defines the *publish-subscribe* dependency between the CoffeeMaker component and the Dispenser components. In Fig. 5(a), when *PIPE* invokes the CoffeeMaker, it gets the recipe data and then pipes it to *SQ1*. *SQ1* then invokes all the Dispenser components so that they dispense different amounts of ingredients according to the piped-in recipe data.[2] Thus the Pipe-Sequencer hierarchy is an *Observer* composite connector (Fig. 5(b)). Of course here *Observer* is used to compose components rather than objects.

As in its object-oriented counterpart, there are two main roles for components composed by an *Observer* composite connector: Publisher and Subscriber. When the Publisher is called, the Subscribers must be notified and must behave accordingly. Like its object-oriented counterpart, an *Observer* composite connector defines the one-to-many dependencies between the Publisher and Subscribers.

In general, a composite composition connector CC can be composed from a set of (basic or composite) composition connectors $C_1, \ldots, C_n$. CC can be used to perform a

[2] In the Observer pattern, the order in which the subscribers are notified is not specified. Here we have chosen a sequential order.

composite composition involving all the components that are composed by $C_1, \ldots, C_n$, but all in a single step. Therefore, composite connectors are patterns, and as such, are much more powerful than their sub-connectors.

Using such connectors can make composition more efficient by reducing the number of (levels of) composition. For example comparing Fig. 5(a), with only basic connectors, and Fig. 5(b), with the Observer composite connector, the level of composition is reduced by 1 in the latter.

Finally, composing connectors into composite ones is clearly one form of connector reuse.

4.2 Constructing Composite Connectors

To construct a composite connector *CC* from a set of inter-connected sub-connectors $C_1, \ldots, C_n$ requires the generation of the correct signature for *CC* as a single connector. *CC* connects different and more connectors (components) than its sub-connectors. In particular its signature is not the same as those of its top-level sub-connector.

For example in Fig. 5, the top-level sub-connector of *Observer* is *PIPE* (Fig. 5(a)). *PIPE* actually connects two components: CoffeeMaker and Comp1. Comp1 is a composite component constructed by the lower level connector *SQ1*. By contrast, the *Observer* in Fig. 5(b) connects CoffeeMaker and all the Dispenser components.

Therefore, to construct a composite connector correctly, we have to take care of its signature, by considering the signatures of its sub-connectors, and their composition structure.

To express the composition structure of a composite constructor, we use the notation $C_1[C_2, C_3]$ recursively to denote a composite connector whose top-level connector C_1 is connected to C_2 and C_3 at the next level down, and so on. Fig. 6 shows a general composite connector (denoted by the shaded box).

This connector can be written as $C1[C2, C3[C4, C5, C6]]$.

Once the composition structure of a composite composition connector has been determined, we can implement the connector by using the implementation of its sub-connectors. For simplicity, we shall assume that all the sub-connectors in a composite composition connector are the basic connectors that we described in Section 3.1. As before, each connector is made up of a signature and code. We represent this as *Connector*(*Sig*, *Code*). In general, the signature of a composite connector is generated from the signatures of all the connectors involved in the composition. Specifically, the signature of a composite connector is the union of the signatures of those connectors,

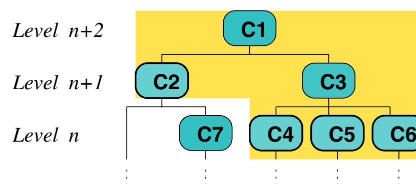

Fig. 6. A general composite connector

including the top-level one, that connect to at least one connector (component) outside of the boundary of the composition. For example, for the composite connector in Fig. 6, its signature is generated from the union of the signatures *Sig1*, *Sig2*, *Sig4*, *Sig5* and *Sig6* of the connectors $C1$, $C2$, $C4$, $C5$, and $C6$. Dependencies between these signatures should be analysed and taken care of while constructing the signature of the composite. In particular, any redundancies resulting from these dependencies should be identified and removed.

The code of the composite connector is implemented by calling its sub-connectors' implementation codes. For instance in Fig. 6, the codes *Code1* ... *Code6* of the sub-connectors $C1 \ldots C6$ already exist, and are used to generate the code for the composite connector, by implementing their dependencies (as specified in the composite connector) as method calls from higher level sub-connectors to the lower level ones. In $C1$'s code *Code1*, $C2$ and $C3$ are specified in the sub-connector list. When $C1$'s `execute` method is called, it invokes every connector in the sub-connector list, i.e. the `execute` methods of $C2$ and $C3$. Since $C3$ is composed from $C4$, $C5$ and $C6$, it further invokes *Code4*, *Code5* and *Code6* to implement the functionalities.

In this way, we construct a composite connector from its sub-connectors. We get a new signature as well as new code for the new connector. The new signature prescribes the usage of the new connector, and typically contains more parameters than the signatures of the sub-connectors. The new connector's code is a collaboration of the sub-connectors' codes performed according to the composition structure of the new connector.

Clearly the composition structure of a composite connector of course determines the nature of the connector. The same set of sub-connectors will result in different composite connectors when composed differently. This is particularly alarming when you consider that composite connectors are patterns. For example, given a connector C, whereas the composite *Pipe*[C,*Sequencer*] is the Observer pattern, as we have seen, by reversing the order of the sub-connectors we get a totally different pattern: *Pipe*[*Sequencer*, C] is the AND-join Pipe pattern (which we will describe below).

Another point worth noting is that in theory, it is possible to build arbitrary composite connectors of unlimited complexity. In practice, some of these connectors may be useless or too hard to use. So there must be some intent when building any composite connector. In other words, useful composite connectors must reflect commonly occurring or recurring control flow patterns, such as the set of workflow control-flow patterns identified in [16].

4.3 Example

Now we show how to construct a commonly occurring workflow control-flow pattern [16], namely the AND-Join Pipe pattern, as a composite connector.

AND-Join Pipe. The intent of the *AND-Join Pipe* composite connector is to allow more than one predecessors in a binary piping composition scheme. It is an "AND" relationship between these predecessors, i.e. only after all the predecessors have been called that the results are gathered and delivered to the successor. This pattern of control can be

achieved by composing the *Pipe* and the *Sequencer* together. This composite connector is equivalent to the *Generalised AND-Join* workflow control-flow pattern [16].

Fig. 7 (a) shows the composition structure of the *AND-Join Pipe* connector. The *Pipe* connects to the *Sequencer* at the predecessor position. The *Sequencer* connects to multiple predecessor connectors (components), i.e. *pred1*, ..., *predN*; and the successor connector (component), i.e. *succ*, is connected to the *Pipe* directly. The dotted line denotes the control-flow path of this connector. It first invokes the *Sequencer* and then, and then the *Sequencer* invokes all the connecting predecessor connectors (components) and returns all the resulting data (denoted with circled D). Finally, the *Pipe* delivers all the results to the successor connector (component) which it takes for its execution.

Fig. 7 (b) shows the signatures of the basic connectors *Pipe* and *Sequencer* and the values they could take for the composition depicted in Fig. 7 (a). As shown in the figure, the signatures require the parameters `tget`, `op` and `par` which correspond to the connectors (components) they connect, the operations to execute through these connectors (on this components), and the required parameters for these executions.

Fig. 7 (c) shows the signature generated for the *AND-Join Pipe*. As can be seen, it differs from those of its sub-connectors, since it connects different and more connectors (components) than its sub-connectors. Note how its signature is not the same as that of its top-level sub-connector (*Pipe*). As we have explained, the signature of the *AND-Join Pipe* is the union of the signatures of all sub-connectors that connect to at least one

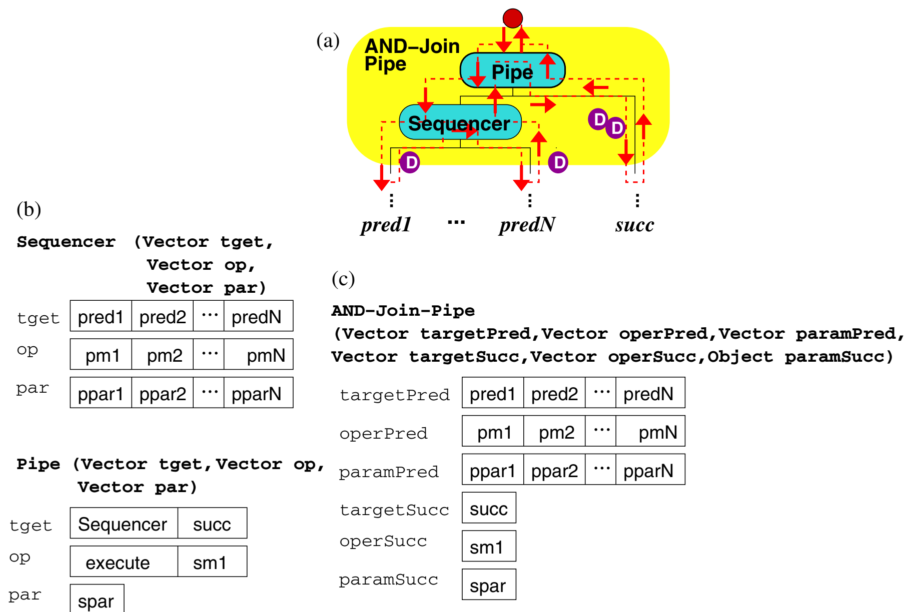

Fig. 7. (a) Composition structure of AND-Join Pipe connector, (b) Signatures of its sub-connectors and (c) Signature of AND-Join Pipe connector.

```
class AND-Join-Pipe extends Connector {
 Sequencer seq;
 AND-Join-Pipe(Vector targetPred, ...){
  ...
  seq = new Sequencer(targetPred, ...);
  ...
 }
 Object execute (){
  ...
  result = seq.execute();
  ...
  result = executeConnector(targetSucc.elementAt(0), ...);
  ...
  return result;
 }
}
```

Fig. 8. Outline of the code for the AND-Join-Pipe composite connector's class

connector (component) outside of the boundary of the composition. Thus, as shown in Fig. 7 (c), the signature of a *AND-Join Pipe* includes those for *Sequencer* and *Pipe*.

In the signature of the AND-Join Pipe, we have removed redundancies arising from the signatures of *Sequencer* and *Pipe*. Notice how the signature elements corresponding to the *Pipe* connector (`targetSucc`, `operSucc` and `paramSucc`) do not include an entry for referring to the *Sequencer* connector. The connection to *Sequencer* is defined in the composition structure of the composite connector and so has been coded in.

The connectors' code of the *AND-Join Pipe* connector is a collaboration of the sub-connectors' codes and, as in basic connectors, it is encapsulated in its `execute` method. Fig. 8 shows an outline of it.

When the connector is created, via its constructor, an instance of a *Sequencer* connector is generated with the corresponding values in the signature, i.e. `targetPred`, `operPred` and `paramPred`. Then, this instance (`seq`) is used in the `execute` method to execute the *Sequencer* by calling its `execute` method. Later, and given that the type of the successor connector is unknown at this point, the execution of the successor connector is carried out by calling the `executeConnector` method.

The process of creating composite connectors can be partially automated by using a graphical tool. We have implemented such a tool. The tool provides a visual way to drag connectors into a composition environment, connect them and generate the skeleton for the resulting connector's class. The skeleton has to be filled in; this is done manually at present. Then the completed connector can be deposited in the tool's repository.

Fig. 9 shows an example of using this tool. On the left hand side, we can see a *Pipe* and a *Sequencer*. These connectors are connected together using a line. The line indicates to the tool that these connectors should be composed to make a composite connector. The tool then generates a skeleton for the composite connector, and the user fills in the skeleton. On the right hand side of Fig. 9, we can see the constructed connector, *AND-Join Pipe* in the connector repository.

Analogously, a *Pipe* is composed with the *Selector* on the left hand side of Fig. 9. The composition result is an *Exclusive OR-Split Pipe* connector. This composite connector models the piping control that allows multiple successors but chooses only one depending on the output value of the predecessor. This connector is constructed from composing a *Selector* to the successor position of a *Pipe*. It behaves like the Exclusive

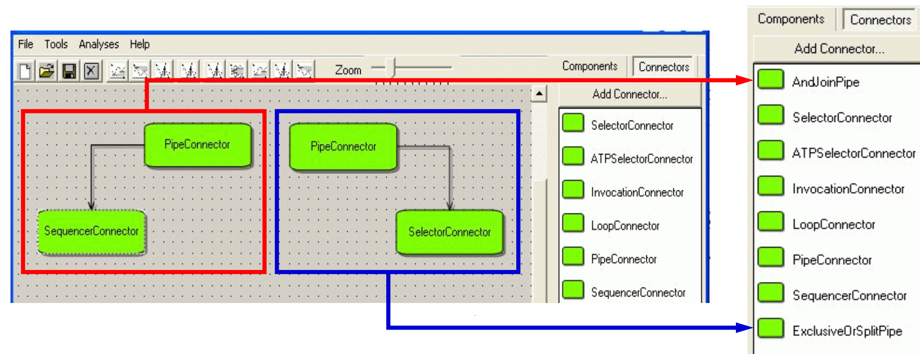

Fig. 9. Building composite connectors by using a graphical tool

OR-Split workflow control-flow pattern [16]. In Fig. 9, the *Exclusive OR-Split Pipe* has also been constructed and put in the connector repository.

An example using these two composite connectors will be shown in next section.

5 Using Composite Connectors in Practice

Having explained how composite connectors are constructed, in this section, we show to use composite connectors to build a complete system. We will use the example of an Automatic Train Protection (ATP) system.

To construct a system from our components and composition connectors, we use an assembler-container tool [9] that we have built. The assembler-container hosts components and connectors and manages their assembly. It takes three main inputs: (a) a set of components; (b) a set of composition connectors; and (c) an XML description of the connector hierarchy of the system. The three inputs are independent from each other. The output of the assembler-container is a run-time system constructed in accordance with the XML description, with the top level connector as an interface to the system.

The assembler-container does not distinguish between basic and composite composition connectors. So we can use our composite connectors to build systems in the assembler-container. As an illustration we will show how the ATP system can be built both with and without composite composition connectors.

The ATP system is located on board a train to ensure safety. The system consists of the following components: Sensor 1, 2 and 3, SensorAggregator, ATPController, Brakes, Alarm, Speedometer and CautionStateProcessor. The sensors are attached to the side of the train and detect information on the track-side signals. Each sensor generates a signal in the range {DANGER, CAUTION, PROCEED}. The overall resulting signal is then sent to the other components. The components must respond to the signal accordingly, e.g. Alarm and Brakes must be enabled when the signal is DANGER.

Using only basic connectors, the ATP system can be built with the architecture shown in Fig. 10.

This architecture consists of 9 components and 13 composition connectors on 6 levels.

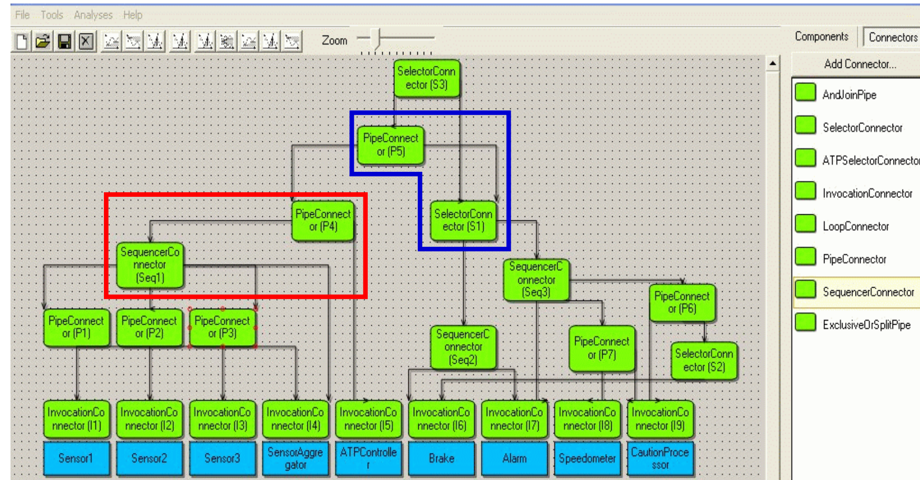

Fig. 10. Automated Train Protection System without composite connectors

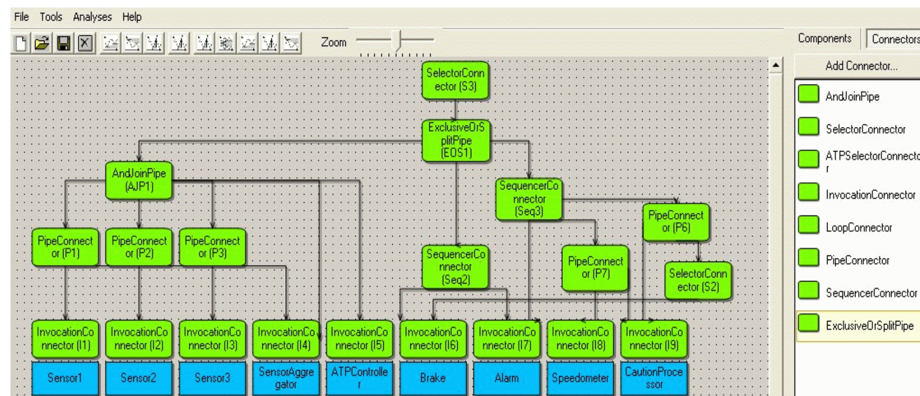

Fig. 11. Automated Train Protection System with composite connectors

Looking at the connector hierarchy in Fig. 10, it is clear that we can compose some basic connectors into composite connectors. The latter are indicated in the figure by two groups of basic connectors encircled by a bold line. These two composite connectors are in fact an *AND-Join Pipe* and an *Exclusive OR-Split Pipe* (Section 4.3).

The graphical tool for building composite connectors (Section 4.3) is integrated with the assembler-container, so we can build the *AND-Join Pipe* and *Exclusive OR-Split Pipe* connectors in the assembler-container, and then use them to build the ATP system.

Using these composite connectors, we can reduce the complexity of the ATP system, and change its architecture to that in Fig. 11.

From the system architecture in Fig. 11 we can see that a composite connector is used in the same manner as the basic ones in hierarchical composition. Also, comparing

Fig. 10 and Fig. 11, we see that using composite connectors reduces the complexity of ATP system by 2 connectors and 1 hierarchy level.

6 Discussion

As pointed out in [15], software systems are getting increasingly complex, and so building them will require more powerful connectors than basic ones such as RPC (remote procedure call). We believe our approach to composite connectors can be used to build suitable connectors. By building composite connectors hierarchically from sub-connectors, we can build composites of arbitrary complexity and functionality.

Our connectors are generic, compositional and reusable. Their genericity and compositionality are demonstrated by the fact that they are control flow patterns. They behave like object-oriented design patterns [6] that coordinate communications between objects, e.g. the Observer pattern, as we saw in Section 4.1. Furthermore, because they coordinate components that do not initiate communication with other components, they correspond even more closely to workflow control-flow patterns [16].

However, in contrast to object-oriented design patterns and workflow control-flow patterns, our composite connectors are reusable as real pieces of implementation. Object-oriented design patterns are generic solutions. The idea behind such a pattern can be used for many applications, but the pattern itself has no generic implementation and has to be coded into every application. A workflow control-flow pattern also does not have any generic implementation. This is because it represents a process, and it is only defined when the workflow (with the activities involved) has been fixed.

Clearly there are object-oriented design patterns that cannot be represented by our composite connectors, namely (i) patterns that do not coordinate communications, (ii) patterns that are specific only to objects, e.g. creational patterns. Conversely, there are object-oriented design patterns that can be represented as a basic connector in our model. For example, the Mediator pattern can be implemented as a *Sequencer* that has been enhanced with an iterator.

Equally, there are many workflow control-flow patterns that cannot be represented by our composite connectors. In particular, those that involve concurrency. We have no concurrency in our model as yet.

7 Conclusion

In this paper we have presented a set of composite composition connectors for component composition, which are ready-to-use for building systems out of reusable components encapsulating computation only. These operators are defined within the context of our component model, and are based on the idea of exogenous connectors.

We have demonstrated that the hierarchical nature of our exogenous composition connectors makes it not only possible, but also easy to generate composite composition connectors. We have demonstrated the use of our connectors for constructing systems by means of an example. Additionally, these composite composition connectors can also be seen as patterns that can be used to perform complicated compositions involving many components all in a single step.

To further enhance its usefulness and efficiency, we plan to extend our set of basic operators to concurrency, so that we get composites able to deal with multi-threading issues, etc.

References

1. Arbab, F.: Reo: A channel-based coordination model for component composition. Mathematical Structures in Computer Science 14(3), 329–366 (2004)
2. Arbab, F.: Abstract behavior types: a foundation model for components and their composition. Sci. Comput. Program. 55(1-3), 3–52 (2005)
3. Böhm, C., Jacopini, G.: Flow diagrams, turing machines and languages with only two formation rules. Communications of the ACM 9(5), 366–371 (1966)
4. Bures, T., Plasil, F.: Composing connectors of elements. Technical Report 2003/3, Dep. of SW Engineering, Charles University, Prague (2003)
5. Bures, T., Plasil, F.: Scalable-element based connectors. In: Ramamoorthy, C.V., Lee, R., Lee, K.W. (eds.) SERA 2003. LNCS, vol. 3026, pp. 198–204. Springer, Heidelberg (2004)
6. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable Object-Oriented Software. Professional Computing Series. Addison-Wesley, Reading (1995)
7. Lau, K.-K., Elizondo, P.V., Wang, Z.: Exogenous connectors for software components. In: Heineman, G.T., Crnkovic, I., Schmidt, H., Stafford, J., Szyperski, C., Wallnau, K. (eds.) Proceedings of 8th Int. SIGSOFT Symposium on Component-based Software Engineering, pp. 90–106. Springer, Heidelberg (2005)
8. Lau, K.-K., Ornaghi, M., Wang, Z.: A software component model and its preliminary formalisation. In: de Boer, F.S., Bonsangue, M.M., Graf, S., de Roever, W.-P. (eds.) FMCO 2005. LNCS, vol. 4111, pp. 1–21. Springer, Heidelberg (2006)
9. Lau, K.-K., Ukis, V.: Automatic control flow generation from software architectures. In: Löwe, W., Südholt, M. (eds.) SC 2006. LNCS, vol. 4089, pp. 323–338. Springer, Heidelberg (2006)
10. Lopes, A., Wermelinger, M., Fiadeiro, J.L.: A compositional approach to connector construction. In: Cerioli, M., Reggio, G. (eds.) WADT 2001. LNCS, vol. 2267, pp. 201–220. Springer, Heidelberg (2002)
11. Medvidovic, N., Taylor, R.N.: A classification and comparison framework for software architecture description languages. Software Engineering 26(1), 70–93 (2000)
12. Le Metayer, D., Nicolas, V.-A., Ridoux, O.: Programs, Properties, and Data: Exploring the Software Development Trilogy. IEEE Software 15(6), 75–81 (1998)
13. Papadopoulos, G.A., Arbab, F.: The Engineering of Large Systems. Advances in Computers 46, 329–400 (1998)
14. Shaw, M., Garlan, D.: Software Architecture: Perspectives on an Emerging Discipline. Prentice-Hall, Englewood Cliffs (1996)
15. Spitznagel, B., Garlan, D.: A compositional approach for constructing connectors. In: WICSA 2001. In: Proceedings of the Working IEEE/IFIP Conference on Software Architecture (August 2001)
16. van der Aalst, W.M.P., ter Hofstede, A.H.M., Kiepuszewski, B., Barros, A.P.: Workflow patterns. In: Distributed and Parallel Databases, pp. 5–51 (2003)