Volume 83, issue 7, July 2010
ISSN 0164-1212
ELSEVIER
The Journal of Systems and Software
Special Issue: SPLC 2008

The Journal of Systems and Software 83 (2010) 1165–1178

Contents lists available at ScienceDirect

The Journal of Systems and Software

journal homepage: www.elsevier.com/locate/jss

A catalogue of component connectors to support development with reuse

Perla Velasco Elizondo [a,*], Kung-Kiu Lau [b]

[a] Centre for Mathematical Research, CIMAT, Guanajuato 36240, Mexico
[b] School of Computer Science, The University of Manchester, Manchester M13 9PL, UK

ARTICLE INFO

Article history:
Received 25 June 2008
Received in revised form 16 July 2009
Accepted 6 January 2010
Available online 14 January 2010

Keywords:
Software components
Connectors
Composition
Adaptation
Development with reuse

ABSTRACT

Component-based development is based on the idea of building software systems by composing pre-existing components. Connectors are the 'glue' for composing components. Therefore, it is important to consider connectors as first-class entities and provide adequate descriptions of them to facilitate their understanding and promote their reuse. We have defined a catalogue of component connectors to support the process of 'development with reuse'. The categories and connector types in the catalogue were obtained through an analysis of the activities involved in this process as well as considering the syntax and semantics of a new component model.

1. Introduction

Component-based development (CBD) aims to develop software systems by *reusing* pre-existing software components rather than coding them entirely from scratch. In this context, there is a general understanding of components as *reusable* building blocks of computation that are *composed* together into larger composites.

The achievement of this goal depends not only on the availability of the components, but also on how the components are assembled together, and what kind of composition mechanisms are used. We believe that current CBD approaches have not yet entirely succeeded in developing different software systems by composing pre-existing software components.

In current CBD approaches (Lau and Wang, 2007), components are either constructed from scratch or, because of the nature of the composition mechanisms used, an integral part of the composition mechanisms themselves. Constructing components from scratch goes against the spirit of composing pre-existing components, which is the basis of CBD. On the other hand, components containing very specific composition information are *highly coupled* with the components with which they communicate, which hinders their reuse for the construction of different systems. Additionally, such components become *more complex* due to the mixing of two basic concerns: *computation* and *composition*. We believe that in CBD, the separation of *computation* from *composition* is a fundamental pre-requisite, as the *reuse* of computation is a key issue. Separating computation from composition means that system specific composition issues are not in components and therefore components can be reused many times for constructing different systems, so that reuse is really maximised.

Thus, in Lau et al. (2005) we introduced an alternative composition approach based on connectors that encapsulate *schemes* to deal with *communication* and *coordination* among components. These connectors are not built into the components, so that they are indeed *first-class* compilation units and the sole medium of communication and coordination among components. In Lau et al. (2007b,a, 2005), we have introduced some basic connector types. In this paper, we further elaborate on these and other new connectors and classify them in a catalogue for supporting the process of *development with reuse* (Sommerville, 2004). The categories and connector types in the catalogue resulted from an analysis of the activities involved in this process, as well as taking into consideration the syntax and semantics of a new component model that we have proposed (Lau et al., 2006).

In Section 2, we explain the generalities of our connectors within the context of the new component model. Next, in Section 3, we introduce the catalogue and justify the proposed connector categories: (i) adaptation connectors and (ii) composition connectors. In Sections 4 and 5, we describe the connector types in (i) and (ii), respectively. In Section 6, we discuss the differences between connectors to support development with reuse and connectors to support *development for reuse* (Sommerville, 2004). In Section 7, we briefly discuss the use of the catalogue of connectors in practice. Finally, in Sections 8 and 9 we discuss and evaluate the material presented in this paper and state the conclusions and future work.

[*] Corresponding author. Tel.: +52 492 998 1529.
E-mail addresses: pvelasco@cimat.mx (P. Velasco Elizondo), kung-kiu@cs.man.ac.uk (K.-K. Lau).

0164-1212/$ - see front matter © 2010 Elsevier Inc. All rights reserved.
doi:10.1016/j.jss.2010.01.008

2. A new view of connectors within a new component model

The concept of *composition* encapsulates the notion of taking two or more components and putting them together in some way. Composition should be *constructive*. That is, it should result in a new entity that has some of the properties of its constituents. However, this view of composition has been neglected in the majority of CBD approaches due to the nature of the composition mechanisms employed. Our connectors fill this gap. By way of contrast, before explaining our connectors, we first analyse the notion of "composition" and the mechanisms used in current CBD approaches.

Current component-based approaches (Lau and Wang, 2007) tend to use composition mechanisms that fall into two main categories: (a) *direct message passing* and (b) *indirect message passing*.

Connection by direct message passing corresponds to *direct method calls*, see Fig. 1a. In general, in the direct message passing scheme, there are two distinct roles: the *sender* and the *receiver* of a message. The identity of the receiver is either statically known to the sender or it is dynamically evaluated at execution time. From the sender's point of view, the identity of the receiver is known *a priori*, but the receiver does not have to know the sender at all. The "send" operation is generally targeted by the sender at a specific set of receivers. When components are connected by direct message passing any *data flow* and *control flow* related to the "composition" is mixed with the *computation* preformed in the individual components. Thus, there is no explicit code for connectors, since messages are "*hard-wired*" into the components; it makes the sender and receiver components tightly coupled with one another. Similarly, there is no explicit code for the resulting "composition".

Remote procedure calls (RPC), method and event delegation are well-known examples of direct message passing schemes. Examples of component models that adopt direct message passing are EJB (EJB Web Page), CCM (CORBA Component Model Web Page), and KobrA (Atkinson et al., 2001).

Connection by indirect message passing is exemplified in Fig. 1b. Here, connectors are separate entities that are defined explicitly. Typically they are glue code or scripts that pass messages between components indirectly. To connect a component to another component a connector is used that, when notified by the former, invokes a method in the latter. In general, when components are connected by indirect message passing, the data flow related to the composition is separated from the computation in the individual components. However, this does not necessarily mean that control flow is separated from computation, since the information passed from and to a component may contain method calls. As a result, components are tightly coupled by connectors. Although they are separate entities, connectors are usually not intended to be defined and deposited in a repository. Rather, they are pieces of code generated for specific sets of components. Therefore, these connectors are not reusable.

Examples of component models that adopt indirect message passing are Koala (Ommering et al., 2000), PECOS (Nierstrasz et al., 2002), JavaBeans (JavaBeans Web Page) and some Architecture Description Languages (ADLs) (Medvidovic and Taylor, 2000; Clements, 1996) such as C2 (The C2 Style Web Page) and ACME (Monroe et al., 1997).

Fig. 1a and b illustrate that, in these composition approaches, the data flow and control flow required to set up the component composition originates in and flows from the components. In other words, components deal not only with computation but also with *communication* (data flow) and *coordination* (control flow). By contrast, in our approach all the communication and coordination originates in and flows from connectors, leaving components to encapsulate only computation. We call these connectors *exogenous connectors* (Lau et al., 2005; Lau and Ornaghi, 2009), because in contrast to existing connectors, they encapsulate all the communication and coordination *outside* the components. Our components represent *loci of computation* in a software system. However, they do not request services from other components. Rather, they perform their provided services only when invoked by connectors, according to the communication and coordination schemes these connectors encapsulate.

As we will describe later, the control schemes encapsulated in our connectors are analogous to either *control structures* that can be found in most programming languages or to well-known *behavioural patterns*. For example, we have a *Selector* connector which selects the execution of the computation of only one component from a set of components, on the basis of the value of a Boolean expression. As can be inferred, the communication and coordination scheme encapsulated by the *Selector* connector is analogous to the *if–then-else* control structure. However, it is important to emphasise that the communication and coordination schemes encapsulated in our connectors exclusively deal with the communication and coordination issues among a set of components, rather than the schemes required by each component to perform its own *computation*.

Fig. 1c shows the connectors in our approach. As can be seen, components do not call methods in other components. Instead, all method calls are initiated and coordinated by the connectors. The round dots denote the origins of the communication and coordination. This is in clear contrast to both Fig. 1a and b where components originate communication and coordination.

In the new component model, software systems are described in terms of two kinds of distinct elements: *components* and *connectors*. Fig. 2 shows a system architecture in such a context. It consists of a hierarchy of connectors ($K_1 - K_5$) representing the system's communication and coordination, sitting on top of com-

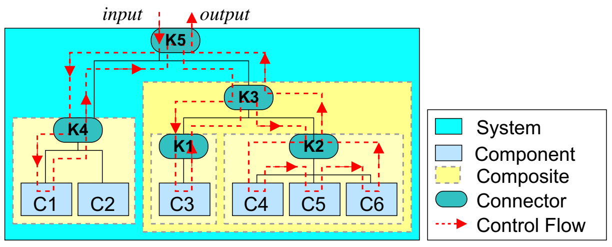


Fig. 2. A system architecture in our approach.

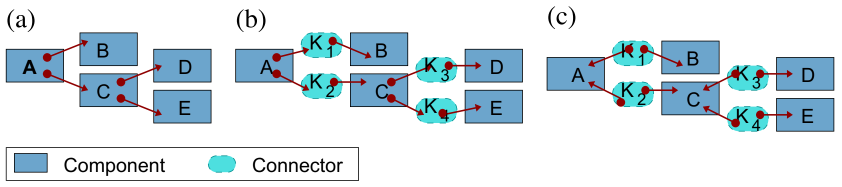


Fig. 1. Component composition via (a) direct message passing, (b) indirect message passing and (c) exogenous connectors.

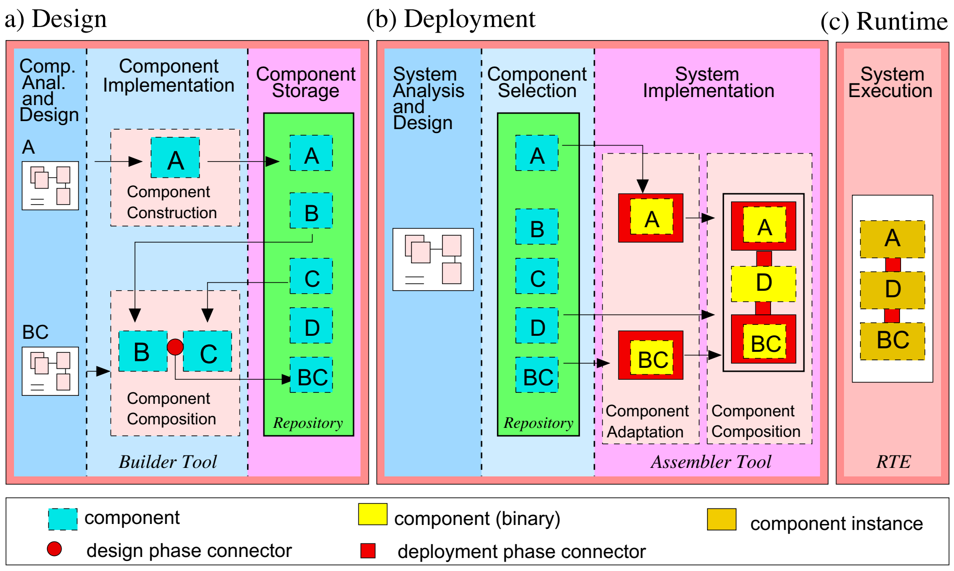


Fig. 3. An idealised component life cycle.

ponents that provide the computation performed by the system. Independently of its type and *arity*, any connector works as a *composition operator* that promotes *compositionality* (Lau et al., 2006); that is, when applied to components it yields another component. The resulting component can in turn be a subject of further composition. This is illustrated in Fig. 2 by the inner dotted boxes which represent composite components.

In our approach, any assembly of components is built in a *hierarchical bottom-up* manner while the execution of the resulting system is carried out *top-down*. A system architecture has a set of possible execution paths, but only one execution path is executed at any one time according to the *input* data and the type of connectors in the hierarchy. The dotted arrows in Fig. 2 show one possible control flow path of the depicted system, which involves the execution of the computation in the components C1, C3 and C4–C6. The *output* generated is given at the end of the system execution via the top-level connector.

Now that the generalities of our connectors and their role in the new component model have been explained, in the following section we describe the proposed catalogue.

3. The proposed catalogue

Our catalogue of connectors is defined within the context of an *idealised component life cycle* (Lau and Wang, 2007). This life cycle considers both the *development for reuse* and *development with reuse* processes, as CBD is concerned with both the development (development for reuse) and consumption (development with reuse) of reusable software components throughout the development lifecycle. The idealised life cycle considers the former processes by defining three main phases: *design*, *deployment* and *runtime*.

Fig. 3a depicts the *design phase* in the idealised component life cycle. The focus of this phase is on *component development*. In the design phase *component developers* generate a set of components, so that these components work as *templates*, i.e. generic skeletons which can be reused for the construction of different systems within an *application domain*. The design phase embodies the process of *development for reuse*; accordingly we have considered in this phase activities such as *component analysis and design*, *component implementation* and *component storage*. Note that, besides constructing *atomic components*,[1] we also construct *composite components* by using *design phase connectors*, e.g. in Fig. 3a B and C are retrieved from a *repository* and later composed into a composite component BC via a *Builder Tool*. Thus, within the *component implementation* activity it makes sense to distinguish between *component construction* and *component composition*.

Fig. 3b depicts the *deployment phase*. In this phase *system developers* assemble *binary* components together by using *deployment phase connectors* to create *ready-to-run* systems. Ideally, deployment phase composition should be carried out on components generated during the design phase. This is convenient because it leads to a scenario where reuse is maximised and components are indeed units of third-party composition. The deployment phase corresponds to the process of *development with reuse* so that, besides *system analysis and design*, *system implementation*, the activities in this phase also include *component selection*.

Finally, Fig. 3c illustrates the *runtime phase* where the constructed system is loaded, instantiated and executed in the runtime environment (RTE). This phase is also part of the process of development with reuse. Currently we have not further elaborated on the activities in this phase. However, it may be desirable to adapt component instances dynamically to re-configure the executable system.

The connectors proposed in our catalogue are meant to be utilised to support the process of development with reuse. Specifically, the *system implementation* activity depicted in Fig. 3b. As introduced before, component-based systems are built by composing pre-existing components retrieved from a repository (preferably via an *assembler tool*). In practice, however, it is common that the composed components often do not fit each other and changes need to be made to eliminate these conflicts. The process of changing the component to use in a particular application is often referred to as *adaptation* (Bosch, 1999). Thus, within the *system implementation* activity we consider not only *component composition* but also *component adaptation*.

By analysing the data flow and control flow required in common component-based applications, a set of useful communication and coordination schemes, that can be defined as specific connec-

[1] We consider an atomic component as the most basic kind of component in the component model.

Table 1
A catalogue of connectors to support development with reuse.

Category	Type
Adaptation connectors	Guard
	Condition-controlled loop
	Counter-controlled loop
	Delay
Composition connectors	Sequencer
	Pipe
	Selector
	Observer
	Chain of responsibility
	Exclusive choice sequencer
	Exclusive choice pipe
	Simple merge sequencer
	Simple merge pipe

tor types, has been identified. The schemes in the set are analogous to either *control structures* that can be found in most programming languages or to *behavioural patterns*.

Table 1 shows the proposed catalogue. As can be seen, the catalogue is organised in two categories: (i) *adaptation connectors* and (ii) *composition connectors*. In the following sections, we describe these categories and the connector types therein.

4. Adaptation connectors

In the context of CBD, system developers perform component adaptation to tackle component mismatches to obtain customised versions suitable for the systems being developed. Component mismatches can be found at different levels, from small syntactic differences to complex semantic ones (Becker et al., 2006, 2004). These mismatches can be fixed by a wide range of component adaptation techniques, e.g. (Becker et al., 2006; Kim and Lee, 2005; Heineman, 2000; Bosch, 1999; Heineman and Ohlenbusch, 1999).

Given the nature of our connectors, we cannot tackle issues related to syntactic mismatches, e.g. operation names, filtering of operations, parameter names, etc. However, we can adapt the components by adding some control to alter the existing functionality in a component via an *adaptation connector*.

An adaptation connector is a *unary* connector. It can be seen as a kind of "prefix" for adding an extra bit of control to a component. The adaptation connector adapts the component in the sense that before any computation takes place inside the component, the execution of the control scheme encapsulated by the connector is executed first.

Table 2 presents the set of adaptation connector types in our catalogue as well as the corresponding descriptions. The set includes the *guard*, *condition-controlled loop*, *counter-controlled loop* and *delay* connectors. Fig. 4 illustrates the behaviour of the resulting assemblies by using a customised UML activity diagram notation, which allows us to focus on the flows of control and data. In this notation, the dotted boxes represent the resulting assemblies. The boxes with the computation label represent the computation in the adapted components. An edge connecting two diagram elements represents the control flow along the direction of the edge. Any data required in the assembly to perform the corresponding computation is denoted as the *input* label. The result generated by the assembly execution is denoted as the *output* label.

Fig. 4a depicts the case of the *guard* adaptation connector. As can be seen, any computation in the adapted component is conditional upon the value of a boolean expression (expr) being *true*.

Fig. 4b and c depict the cases of the *condition-controlled loop* and *counter-controlled loop*, respectively. These two connectors provide a *looping control scheme*. In the case of the *condition-controlled loop* connector, the iterative execution of computation in a component is performed until a boolean expression is not satisfied. In the case of the *counter-controlled loop* the computation is executed repeatedly a fixed number of times. Once the expressions that allow the iterations are not satisfied, no further execution is performed, which is denoted by the termination symbol in Fig. 4b and c.

Finally, the *Delay* connector, depicted in Fig. 5c, delays the execution of the computation in the adapted component for a specified period of time.

Now that we have described the set of adaptation connectors, in the following section we present the types defined in the composition connector category.

Table 2
Set of adaptation connectors.

Type	Description
Guard	Used to 'guard' the execution of the computation in the adapted component according to the evaluation of a Boolean expression
Condition-controlled loop	Used to repeat the execution of the computation in the adapted component according to the evaluation of a Boolean expression
Counter-controlled loop	Used to repeat the execution of the computation in the adapted component a specific number of times
Delay	Used to delay the execution of the computation in the adapted component a specific period of time

5. Composition connectors

As the name suggests, composition connectors are n-*ary* connectors used to support *component composition*. Table 3 presents a more detailed description of some of the composition connectors introduced in Table 1. We call them *basic* composition connectors, to differentiate them from *composite* composition connectors, which we will explain in Section 5.1.

Fig. 5 illustrates the communication and coordination scheme of each one of these composition connectors. As before, the dotted boxes represent the resulting assemblies and the computation *i* boxes represent the computation in the composed components. An edge connecting two diagram elements represents the control flow along the direction of the edge. Any data required in the

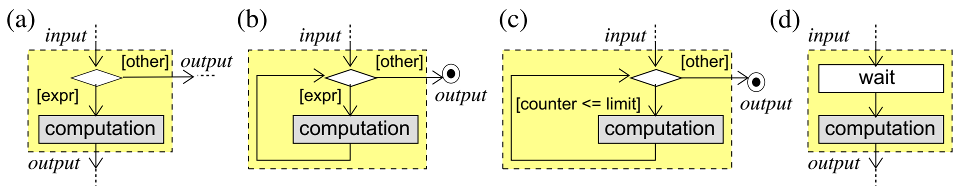


Fig. 4. Behaviour of assemblies resulting from (a) *Guard*, (b) *condition-controlled loop*, (c) *counter-controlled loop* and (d) *delay* connectors.

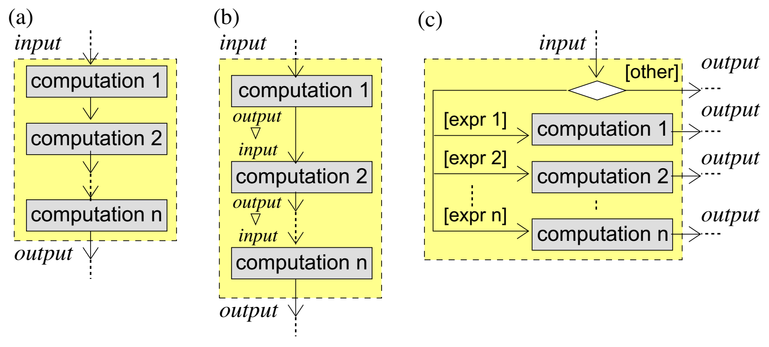


Fig. 5. Behaviour of assemblies resulting from the (a) *sequencer*, (b) *pipe* and (c) *selector* connectors.

Table 3
Basic composition connectors.

Type	Description
Sequencer (SQ)	Provides a composition scheme where the computation in the composed components is executed sequentially one after another
Pipe (P)	Provides a composition scheme where the computation in the composed components is executed sequentially one after another and the output of an execution is the input of the next one and so forth
Selector (SL)	Provides a composition scheme where the computation in only one of the composed components is executed based on the evaluation of a boolean expression

assembly to perform the corresponding computation is denoted as the *input* label. The result generated by the assembly execution is denoted as the *output* label.

Both the *Sequencer* and *Pipe* composition connectors can be used to compose a set of two or more components so that the execution of each one of them is carried out in a *sequential* order, see Fig. 5a and b. In the case of the *pipe* connector, it also models internal data communication among the composed units, so that the *output* generated by a component's execution becomes the *input* to the next one in the chain.

The *selector* composition connector, Fig. 5c, corresponds to the *branching* control scheme. Thus, it can be utilised to compose a set of two or more components so that the execution of only one of them is carried out, the selection being based on the evaluation of a boolean expression (expr *i* in Fig. 5c).

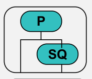
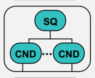
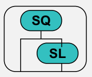
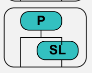
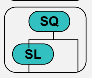
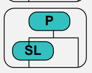
5.1. Composite connectors

In our approach a set of *adjacent* connectors in a system hierarchy can be regarded as a single *composite connector*, e.g. the connectors $K_1 - K_3$ in Fig. 2. Composite connectors are complex connection mechanisms specifically designed to facilitate the communication among the components in a system. While basic connectors such as the *sequencer*, *pipe* and *selector* provide only one type of communication and coordination scheme, composite connectors combine many types. Composite connectors reduce the levels and complexity in system architecture so that the composition is more efficient.

In software engineering, *patterns* are understood as valuable architectural design aids applicable to recurring problems in specific design contexts. Although it is difficult to classify patterns, it is generally accepted that architectural patterns (Kuchana, 2004), design patterns (Gamma et al., 1995), and workflow patterns (Russell et al., 2006) exist. Some of these patterns focus particularly on *behavioural* issues. That is, they are specifically concerned with communication between the participating units. Therefore, *behavioural patterns* provide suitable guidance for creating useful composite connectors.

We have defined the following composite composition connectors: *observer*, *chain of responsibility*, *exclusive choice sequencer*, *exclusive choice pipe*, *simple merge sequencer* and *simple merge pipe*. Table 4 describes them and shows their internal structure. For simplicity, the names given to these connectors are the same as the names of the patterns to which they are analogous. The *observer* and *chain of responsibility* encapsulate a communication and coor-

Table 4
Composite composition connectors.

Type	Description	Structure
Observer	Provides a composition mechanism where once the computation in the "publisher" component has been preformed, the computation in a set of "subscribers" components is executed sequentially	
Chain of responsibility	Provides a composition mechanism where more than one component in set can handle a request for computation	
Exclusive choice sequencer	Provides a composition mechanism where once the computation in a "predecessor" component has been preformed, the computation of only one component ii a set of "successor" components is executed	
Exclusive choice pipe	A version of the exclusive choice sequencer with internal data communication among the "predecessor" and the "successor" components	
Simple merge sequencer	Provides a composition mechanism where once the computation in only one component in a set of "predecessor" components has been preformed, the computation in a "successor" component is executed	
Simple merge pipe (SMP)	A version of the simple merge sequencer with internal data communication between the "predecessor" and the "successor" component	

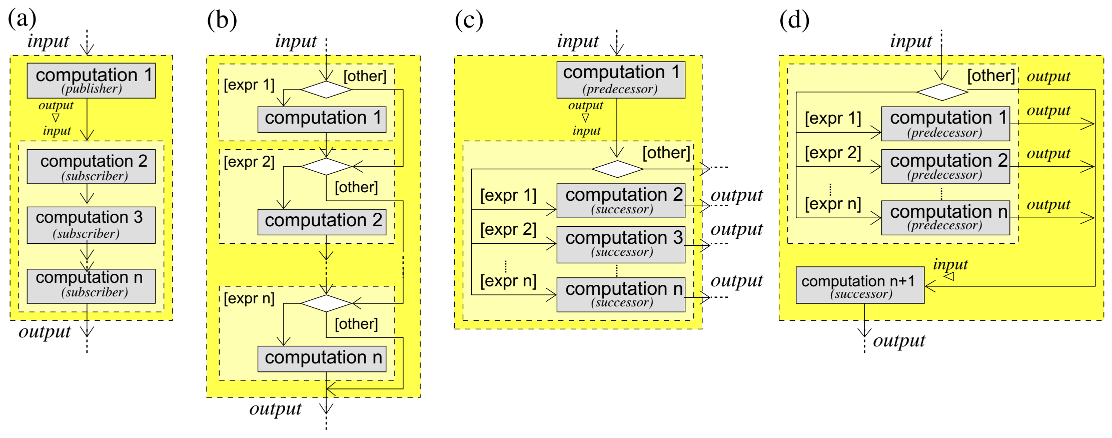

(a)
input
computation 1 (publisher)
output
input
computation 2 (subscriber)
computation 3 (subscriber)
computation n (subscriber)
output
(b)
input
[expr 1]
[other]
computation 1
[expr 2]
[other]
computation 2
[expr n]
[other]
computation n
output
(c)
input
computation 1 (predecessor)
output
input
[other]
[expr 1]
computation 2 (successor)
output
[expr 2]
computation 3 (successor)
output
[expr n]
computation n (successor)
output
(d)
input
[other]
output
[expr 1]
computation 1 (predecessor)
output
[expr 2]
computation 2 (predecessor)
output
[expr n]
computation n (predecessor)
output
input
computation n+1 (successor)
output

Fig. 6. Behaviour of assemblies resulting from the (a) *observer*, (b) *chain of responsibility*, (c) *exclusive choice pipe* and (d) *simple merge pipe* connectors.

dination scheme analogous to the *observer* and *chain of responsibility design patterns*, respectively. The *exclusive choice sequencer*, *exclusive choice pipe* are analogous to the *exclusive choice workflow control flow pattern*. Lastly, the *simple merge sequencer* and *simple merge pipe* are analogous to the *simple merge workflow control flow pattern*. Note that, although all these composite composition connectors are created from arrangements of the basic types, the manner in which these arrangements are defined is crucial for achieving the desired control scheme.

Adopting the usual notation, Fig. 6 illustrates the communication and coordination schemes of some of the connectors in Table 4. Fig. 6a depicts the behaviour of the resulting unit after applying the *observer* connector to a set of components. First, the execution of the computation in the *publisher* component is executed. After that, the *output* generated by this execution becomes the *input* for all the computation in each one of the *n subscriber* components, which are executed sequentially.

Fig. 6b traces the behaviour of the resulting assembly after applying the *chain of responsibility* connector. As can be seen, each one of the components in a chain is "asked" to perform certain computation via the *guard* connectors. Thus, more than one component may perform a computation.

Fig. 6c traces the behaviour of the resulting assembly after applying the *exclusive choice pipe*. Given its internal structure, after the computation in the *predecessor* component has been performed, the generated *output* is passed as *input* data for the computation of only one component in a set of *successor* components. As can be inferred the *exclusive choice sequencer* behaves similarly except for the fact that there is no internal data communication between the *predecessor* and the selected *successor*.

Finally, Fig. 6d depicts the behaviour of an assembly generated via a *simple merge pipe*. First, the execution of the computation in only one component in a set of *predecessor* components is executed. After that, the *output* generated by this execution becomes the *input* for the computation to perform in a *successor* component. Similarly, the behaviour of the *simple merge sequencer* differs from the *simple merge pipe* connector in that there is no internal data communication between the selected *predecessor* and the *successor*.

After describing all the connectors in each one of the categories of the catalogue, in the following section we discuss what differentiates them from the connectors in other phases.

6. Distinguishing between development with reuse and development for reuse connectors

As explained in Section 3, the connectors in our catalogue are meant to be utilised to support the process of *development with reuse* – which maps to the *deployment phase* in the idealised component life cycle. Because our connectors encapsulate communication and coordination schemes, most of these connectors can also be used to support composite component construction during the process of *development for reuse* – which maps to the *design phase* in the idealised component life cycle. However we identify some key aspects that, according to the aim of these processes, cause fundamental differences between the connectors for supporting the process of development for reuse and those for supporting the process of development with reuse. Table 5 summarises these differences.

Table 5
Differences between connectors for 'development for reuse' and those for development with reuse'.

	Development for reuse	Development with reuse
Ideal life cycle phase	Design phase	Deployment phase
Assembly type	Component template + design contracts + deployment contracts + data place-holders	Deployed system
Assembly format	Source or binary, binary	Binary
Connectors	Pre-defined coarse-grain communication and coordination schemes	Pre-defined fine-grain communication and coordination schemes

In the process of development for reuse, any *assembly type* corresponds to a component *template* together with a set of *design* and *deployment contracts*[2] as well as *data place-holders*. A resulting assembly constructed within this process cannot be immediately used on its own; it needs to be deployed into a specific execution environment. In the process of development with reuse, any assembly corresponds to a *deployed system* which is *ready-to-execute*. It defines a fixed implementation of a required behaviour by calling specific component services in the context of a specific environment (e.g. access permissions, databases location, etc.), that has been fully specified via common mechanisms such as system constructors and deployment descriptors.

In the process of development for reuse the *assembly format* is either *source code* or *binary*. In the process of development with reuse it can only be *binary* as an assembly is a *ready-to-execute* unit.

In both the development for reuse and development with reuse processes, connectors are *pre-defined* communication and coordi-

[2] Both design and deployment contracts are used to better specify components.

nation schemes defined within the context of a composition theory (Lau et al., 2006). However, as the components must be sufficiently general to cover the different aspects of their use, connectors supporting the process of development for reuse encapsulate "*coarse-grain*" communication and coordination schemes. These *coarse-grain* schemes are good for supporting a style of composition where the specification of the specific manner in which these assemblies are used is deferred to either the deployment or to the runtime phase of the idealised component life cycle. For example, consider the situation in which the atomic components B and C offer a set of services common for the construction of sales applications, e.g. billing and shipping services. It is sensible and convenient to compose these components together in a BC composite component, as depicted in Fig. 3a. As a system template, the BC assembly can be reused for the construction of different systems in the same domain. Thus in the deployment phase, it should offer a means to invoke any of the operations in its constituents, as well as a means to specify the deployment information of the particular system under construction, e.g. the location of the clients and shipping fares databases.

In contrast, connectors supporting the process of development with reuse encapsulate "*fine-grain*" communication and coordination schemes. That is, these connectors require the precise information to invoke the right operations in the right components for the particular system considering also the characteristics of the execution environment in which the system will be deployed and executed.

7. Putting the catalogue into practice

In some preliminary work, we have already illustrated the use of some of the connectors in our catalogue to support development with reuse (Lau et al., 2007b,a, 2005). Next we briefly describe how we have done it. The complete details can be found in (Velasco Elizondo, 2008).

7.1. Components

As stated before, our connectors are meant to be utilised on reusable components as they support the process of development with reuse. Thus, all these components have been developed during the process of development for reuse according to the semantics of our component model.

Each component is offered as a *binary* file which corresponds to an *implementation* and relates to an *interface*. The component's implementation is a set of Java classes implementing a set of methods. Only the public methods correspond to the services provided by the component. The component's interface, which is written in the form of Java annotations, contains a description of the services provided by the component. The information in the interface together with other component classes' metadata is read by the connectors, via *reflection techniques* (Java TM 2 Platform Standard Ed. 5.0), to discover and to support the invocation of the services provided by a component.

7.2. Connectors

As introduced in Section 2, our connectors allow system construction in a *hierarchical bottom-up* approach. Thus, a system is generated from the application of composition and/or adaptation connectors to components iteratively according to some restrictions on their *arity*. As described in previous sections, composition connectors are always utilised to compose a set of $n > 2$ components while adaptation connectors operate on $n = 1$ component.

Table 6
Properties a connector can define.

Name	Description
Accept	Set of operations that can be invoked
Execute	Set of operations to execute in a particular execution
Iterator	Order in which the operations are executed
Iteration mode	Restrictions on the order in which the operations are executed, e.g. all operations, some operations, an operation can be executed twice, etc.
Condition	Boolean expression utilised by some connector types, e.g. Selector
Loop test	Time in which a boolean expression is evaluated, e.g. pre, post

To enable their *fine-grain* "tuning", the proposed connectors are *customisable*. The customisation is specified via a set of *properties* that the connector defines. Properties have already been utilised in other component-based development approaches to specify the fine details of a connector, e.g. (The C2 Style Web Page; UniCon Web Page). Each connector defines a number of properties. However, some properties may make sense only in the context of a particular connector type, e.g. a *sequencer* connector does not require the specification of a boolean expression as a *selector* connector does. All the proposed properties concern only *functional* issues of the communication and coordination scheme supported by a connector type. For each property there is a specification that defines the valid value(s) it can take. Table 6 shows the list of connectors' properties utilised in our approach.

Like components, connectors were implemented as a set of Java classes so that they correspond to *identifiable compilation units* and therefore can be *stored* in a repository. Each connector class inherits the superclass Connector and defines a *constructor* to instantiate it, and an *execute* method (originally declared in the Execute interface) that implements its corresponding communication and coordination scheme, see Fig. 7. As each connector has a different semantics, the *execute* method is overridden in each one of the corresponding subclasses according to the required logic. To support its *fine-grain* tuning, each connector's *constructor* and *execute* method are parametrisable with respect to the set of *properties* it defines. Such a parametrisation allows the connector to be *reused* many times for different developments.

We have also defined the class System (Fig. 7), which defines a valid composition. This class holds a reference to a connector, which represents the top-level one.

Thus, to build a system, the system developer needs to make use of all the classes depicted in Fig. 7 to create a *new* class, which defines it. The new class must extend System as well as declare a *constructor* and a *run* method. The *constructor* must contain the code for setting up all the required levels of composition and the *run* method must contain a call to the top-level connector's *execute* method to allow the subsystem's execution.

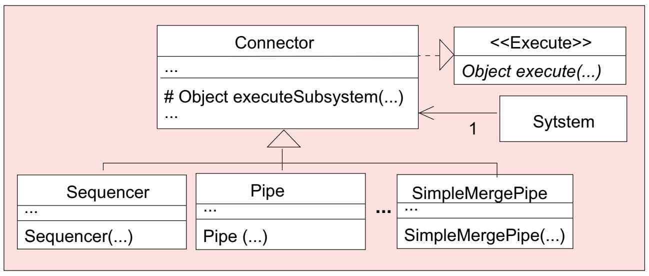
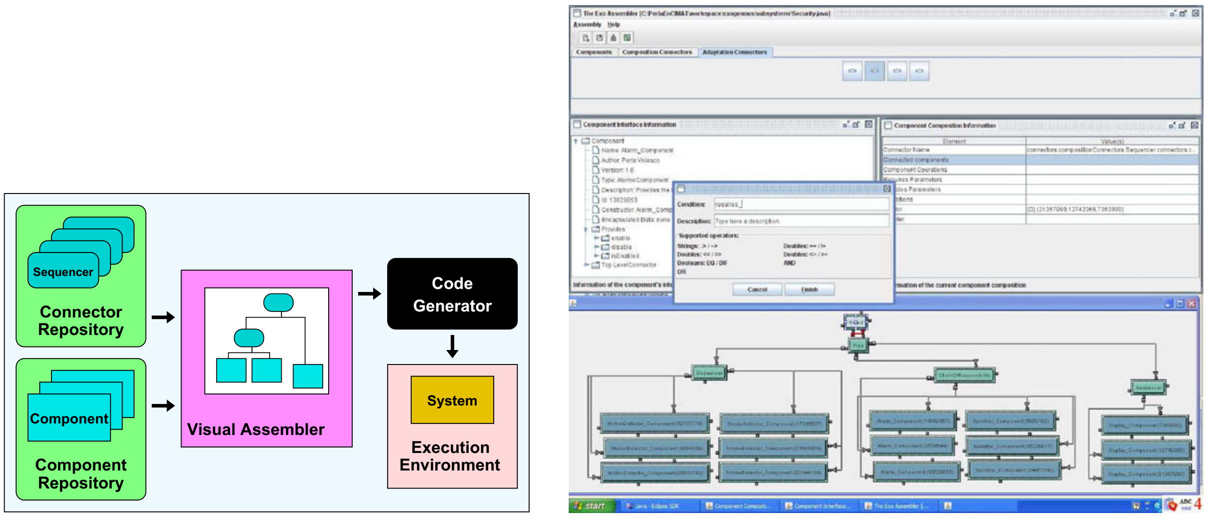


Fig. 7. Class diagram of our connectors' implementation.



(a) (b)

Fig. 8. (a) The architecture and (b) a screenshot of the development tool.

The *execute* method of any deployment phase connector contains a call to the executeSubsystem method defined in the Connector superclass (see Fig. 7). In the executeSubsystem method, the *hierarchical top-down execution* of the system is supported by the following steps: (i) getting the top-level connector of the element, (ii) identifying the top-level connector subtype and (iii) calling the execute method of the corresponding subtype. These steps are repeated for all the subsystems in a hierarchy until the lowest level of the system architecture.

At the lowest level of the architecture, reflection techniques are utilised to dynamically invoke components' operations, as depicted in Fig. 1c. Particularly, we utilise the `invoke` method, which is provided by the class `Method` in the `java.lang.reflect` package.

7.3. System construction

To support automated system development, we have developed a visual tool (Velasco Elizondo, 2009). The architecture of the development tool is depicted in Fig. 8a. As can be seen it consists of five main elements:

– A *component repository*.
– A *connector repository*.
– A *visual assembler*.
– A *code generator*.
– An *execution environment*.

The tool supports development with reuse by dragging, dropping and connecting pre-existing components and pre-existing connectors from the *component* and *connector* repositories into the *visual assembler*. Our tool also uses reflection techniques to perform structural introspection on components' binaries and their interfaces, to retrieve the information about the services that the components offer. Every time a link between a component and a connector is created, the *fine-grain* specification of such a connection is carried out via a composition wizard. According to the semantics of the chosen connector and the information known about the selected component, the wizard asks the system developer to provide specific data according to a pre-defined syntax. All this data is used by the tool to set the values of connectors' properties. The tool performs the corresponding syntactic and semantic checks to ensure the proper definition of an assembly. Fig. 8b shows a screenshot depicting the latest version of the tool used to build the security control system, which we will explain later in Section 9.

Once an architecture is defined by the system developer, the *code generator* generates the source code and the binary file of the new Java class representing the system. Making a system a new class allows us to generate a binary that can be packaged as a named, versioned, shippable and deployable unit. The final system is meant to be deployed and eventually executed on the *execution environment*, which is a JVM.

7.4. Example

In order to show the use of our connectors in practice, in this section we will elaborate on the construction of a simple security control system. The security system's functionality is to sense and interpret all status information, to maintain a display showing current status information, as well as to activate an alarm and call the emergency services (e.g. fire, police, etc.) in the event of an emergency. In order to do that, the security system consists of a number of devices. All the devices are controlled by components. There are two main sensing devices: a motion detector and a smoke detector (SD). According to the running mode of the security system, only one of the two types of the motion detectors must work. A traditional motion detector (MD) is utilised during day time and a infrared motion detector (IRD) is utilised otherwise. There are also one sprinkler (S), one alarm (A), one display (D) and one call manager component (CM).

For simplicity, in this example we use components that do not have complex behaviour. However, we could use components implementing more sophisticated tasks as long as they do not include calls to methods in other components.[3]

Figs. 9 and 10 show the architecture and the behaviour of such a system in the usual notation. In the architecture it is possible to identify three main subsystems: (i) one for sensing the detectors,

[3] For example http://www.cimat.mx/pvelasco/exo/exotool_en.html shows some systems implemented with our connectors and components with more complex behaviour.

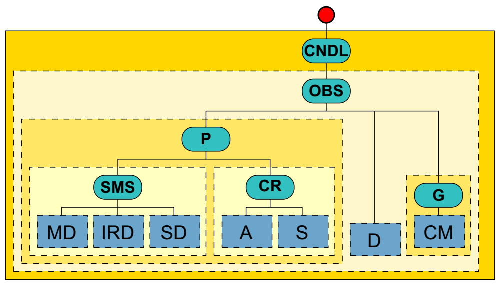


Fig. 9. Architecture of a simple security control system.

(ii) one for firing the alarm and the sprinkler (if necessary) and (iii) one for maintaining the display and call the emergency services in the event of an emergency.

The subsystem responsible for sensing the status of all the detectors has been constructed via a *simple merge sequencer* connector SMS, which composes all the detectors' instances (i.e. MD, IRD and SD). As depicted in Fig. 10, this subsystem allows the invocation of the corresponding operations to read *all* the detectors' status, i.e. readStatus in any of the motion detectors and readState in the smoke detector. Note that the *simple merge sequencer* connector SMS is utilised in this assembly to control the execution of the computation in only one of the motion detector components depending on the operation mode of the system, e.g. ($m = 1$). The results generated by these executions are stored in the variables *md* and *sd*, which denote the states of the motion detector and smoke detector, respectively.

The assembly representing the subsystem responsible for firing the alarm and the sprinkler in the event of an emergency has been generated via a *chain of responsibility* connector CR, see Fig. 9. The connector CR allows firing *both* the alarm and sprinkler (via the activate and start operations) *if* it is required. As can be seen in Fig. 10, in our system we assume that a status value equal to "1" indicates that a detector found some motion/smoke, e.g. ($md = 1$). The activate/start operations return a result that is stored into the variables *as* and *ss*, which denote the alarm's and the sprinkler's states, respectively.

To integrate the aforementioned subsystems and allow the data communication among them, a *pipe* connector P is used. Note that this composition allows us to define a *publisher* component for the third subsystem.

The third subsystem allows displaying current status information on the display and call the emergency services if necessary. For this purpose an *observer* connector OBS is used. First, the execution of the computation in the *publisher* component is executed. After that, the output generated by this execution (i.e. the alarm's and the sprinkler's states) becomes the input for all the computation in each one of the *subscriber* components, which in this case are the display component (D) and the manager component (CM). Note that the execution of the method make call is conditional upon the values of the alarm and the sprinkler's states, i.e. ($as = 1$ or $ss = 1$). That is, if any of them is activated then the call manager component will call the emergency services. As can be seen in Figs. 9 and 10 this behaviour is modelled via the *guard* adaptation connector.

The top-level connector of the security system is a *condition-controlled loop* CNDL to allow it to execute in an endless loop, see Fig. 10.

Program 1 shows an outline of the code generated for the security system via our tool. As defined in the type system presented in Section 7.2, a (sub)system is constructed from *components*, other *subsystems* and *connectors*. In the example code, the constructor `SecuritySystem` and the `run` methods are the mechanisms of *instantiation* and *execution*, respectively. The logic of the constructor involves initiating the assembly's constituents and setting up all the levels of required composition. The composition usually involves several levels. The top level connector is denoted by the variable `conn` that corresponds to a *condition-controlled loop* connector.

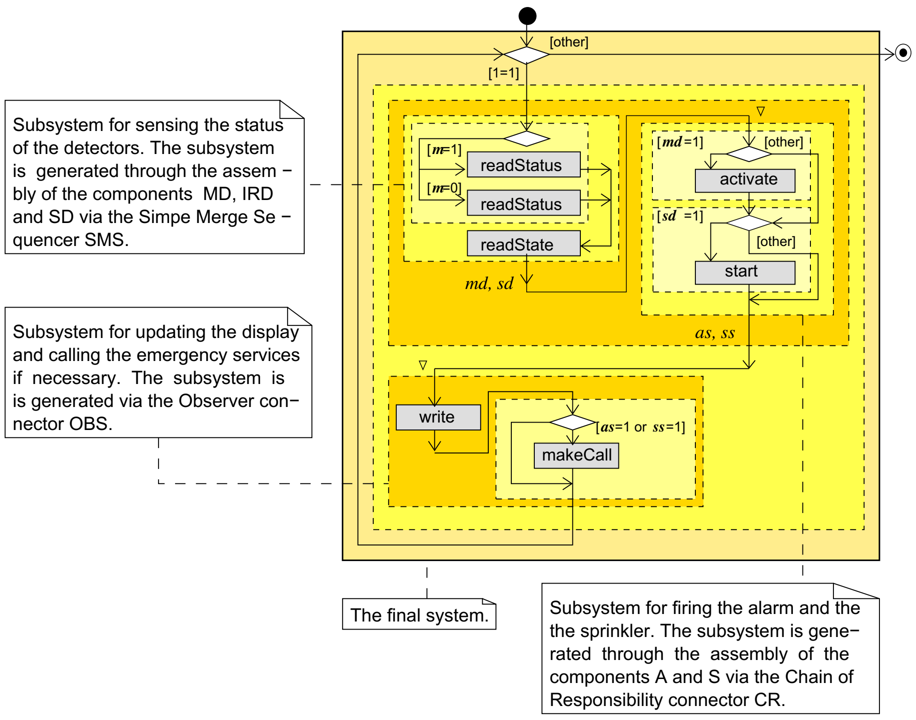

[other]
[1=1]
[m=1]
[m=0]
readStatus
readStatus
readState
md, sd
[md =1]
[other]
activate
[sd =1]
[other]
start
as, ss
write
[as=1 or ss=1]
makeCall
Subsystem for sensing the status of the detectors. The subsystem is generated through the assembly of the components MD, IRD and SD via the Simpe Merge Sequencer SMS.
Subsystem for updating the display and calling the emergency services if necessary. The subsystem is is generated via the Observer connector OBS.
The final system.
Subsystem for firing the alarm and the the sprinkler. The subsystem is generated through the assembly of the components A and S via the Chain of Responsibility connector CR.

Fig. 10. Behaviour of a simple security system.

Program 1. *(Outline of the generated code for the security system)*

```
public class SecuritySystem extends Subsystem {
  private ConditionLoop conn = null;
  public SecuritySystem(...) {
    ...
    /* C O M P O N E N T S C R E A T I O N */
    MotionDetector_Component cmp_3768523 = new
    MotionDetector_Component(...);
    ...
    Display_Component cmp_85918460 = new
    Display_Component(...);
    /* A S S E M B L Y D E F I N I T I O N */
    // ASSEMBLY 1
    condisArray.add("requires_IP_mode == 1");
    targetArray.add(cmp_3768523);
    opersArray.add("readStatus");
    reqParamsArray.add(Constants.NO_PARAMETER);
    provParamsArray.add("provides_CR_md");
    ...
    SMSequencer conn_75938593 = new
    SMSequencer(condisArray, targetArray,...);
    ...
    // ASSEMBLY 2
    condisArray.add("requires_CR_md == 1");
    targetArray.add(cmp_23447542);
    opersArray.add("activate");
    reqParamsArray.add("requires_CR_md");
    provParamsArray.add("provides_CR_as");
    ...
    ChainOfResponsability conn_47880922 = new
    ChainOfResponsability(condisArray,...);
    ...
    // ASSEMBLY 3
    targetArray.add(conn_75938593);
    opersArray.add(Constants.NO_OPERATION);
    reqParamsArray.add("requires_IP_mode == 1");
    provParamsArray.add("provides_CR_md,
    provides_CR_sd,");
    ...
    provParamsArray.add("provides_CR_as,
    provides_CR_ss");
    Pipe conn_76928911 = new Pipe(targetArray,
    opersArray, reqParamsArray,...);
    ...
    // ASSEMBLY 6
    targetArray.add(conn_76928911);
    opersArray.add(Constants.NO_OPERATION);
    reqParamsArray.add("requires_IP_mode == 1");
    provParamsArray.add("provides_CR_as,
    provides_CR_ss,...");
    ...
    Observer conn_60175704 = new
    Observer(targetArray, opersArray,
    reqParamsArray,...);
    // SETTING THE TOP-LEVEL CONNECTOR...
    targetArray.add(conn_60175704);
    ...
    conn = new ConditionLoop(condisArray,
    targetArray, opersArray, reqParamsArray,...);
    setTopLevelConnector(conn);
  }
  public Object run()
    return conn.execute();
  }
}
```

We have defined a set of rules for interface generation according to the semantics of each connector. The information generated via these rules is added in the form of Java annotations into the resulting systems' interfaces. Thus, any defined assembly, if required, can be reused separately for further composition.

8. Discussion and evaluation

In this section we discuss and evaluate our connector catalogue; in particular with respect to (i) pre-defined connectors in component-based development approaches, (ii) patterns as mechanisms for composition, (iii) desirable characteristics of composition and adaptation mechanisms, (iv) the scope of the connectors in the catalogue and (v) the characteristics of our development approach and the generated systems.

8.1. Pre-defined connectors in component-based development approaches

Within the context of software architecture (Bass et al., 2003; Shaw and Garlan, 1996), some ADLs come with a set of *pre-defined connector types*. Thus, a system designer can use and further specify them independently of the components with which they will be used. For evaluation purposes, we focus on this kind of ADLs. Besides that, we take into consideration those ADLs that incorporate some support for code generation to evaluate whether connectors are *traceable* and whether they preserve their *first-class* status at implementation stage. Table 7 lists some connector catalogues in representative ADLs: UniCon (UniCon Web Page), C2 (The C2 Style Web Page) and CLARA (Durand and Déplanche, 1999). UniCon defines six connector types that correspond to the common communication primitives supported by the underlying language or operating system, e.g. RPC, global data, real time scheduler, etc.; C2 comes with a bus connector with a number of filtering and broadcast policies for sending and receiving messages; CLARA offers a set of pre-defined connectors to tackle synchronous and asynchronous event signalling and message passing mechanisms. Our connectors are also listed in the last row.

As Table 7 shows, at design-stage all the connectors in these catalogues can be reused in different system designs because they support some sort of parametrisation. However, at implementation-stage none of the connectors in these catalogues are first-class entities and reusable.

Traceability is a key issue when designing, implementing and maintaining systems (Gast, 2008; Murta et al., 2006; Richardson and Green, 2004; Streitferdt, 2001). In our context, we consider traceability from the perspective of providing traceability support between architectural constructs (i.e. components and connectors) and their implementations in accordance with well-defined mappings. In all ADLs and in our approach such mappings exist. However, the "intrusive" nature of connector mappings in these ADLs results in non-first-class and non-reusable connectors. To illustrate it, consider the bus connector in the C2 ADL. As the underlying communication mechanisms of this connector are events, the defined mappings result in component implementations that also include connectors' code, e.g. the code for defining the specific events a component raise and listen to. Because component implementations include connector implementations, connectors do not correspond to identifiable compilation units anymore, i.e. they do not preserve their first-class status. Therefore, they cannot be stored and reused many times for different developments. In contrast we have demonstrated that, by defining exogenous and parametrised connector implementations, we can tackle these two issues. In the system depicted in Fig. 9, the same *sequencer* connector imple-

Table 7
Comparison of some connector catalogues.

	Design-stage	Implementation-stage		
	Reusable	Traceable	First-class	Reusable
UniCon	√	√	×	×
C2	√	√	×	×
CLARA	√	√	×	×
Our catalogue	√	√	√	√

mentation was (re)utilised to create the two different connector instances SQ1 and SQ2.

Similarly, as they are based on (R)PC and events, the same limitations apply to the pre-defined connectors in work such as SOFA (Bures, 2006), CoCo (Tansalarak and Claypool, 2005) and that discussed by Spitznagel (2004) and Schreiner and Göschka (2007).

8.2. Patterns as mechanisms for component composition

The idea of using *patterns* as mechanisms to support component composition has been already explored. For example, Wydaeghe and Vanderperren developed the idea of *composition patterns* in (Wydaeghe and Vanderperren, 2001). Composition patterns describe interactions between a set of roles, which are fulfilled by components. Both components and interaction patterns are documented by using an extended UML sequence diagram notation. This notation describes a specific usage scenario in terms of a set of participants, the messages they interchange and a set of sequence diagram control blocks such as OPT, ALT and LOOP. A component may be utilised in a composition pattern if its sequence diagram matches a part of that of the composition pattern. Then, a composition is possible if by somehow joining the sequence diagrams of all the participating components it is possible to get the sequence diagram of the composition pattern.

Similarly, the idea of composing fragments of control into larger blocks, in some cases analogous to patterns, has been discussed in several communities. Nevertheless we want to highlight two main aspects that make our approach different from the work in these communities.

Firstly, our composite connectors are real pieces of *generic implementation* that can be stored in a repository. Therefore, the reuse of these connectors is maximised in contrast to the work in these communities. While in theory the idea of a pattern is that it can be used for developing many applications, in practice the pattern itself has no generic implementation and has to be coded into every application. In our case, the same connector template can be utilised to create two different subsystems despite the fact that they differ in the number of participating components and in the messages/data they exchange. This is not possible in Wydaeghe and Vanderperren's work, for example.

Secondly, because our connectors work as operators for components, they support a hierarchical composition approach to produce either composite components or connectors. This is different from related work on composing control fragments or services. For example, within the context of web services, our connectors can define workflow and can produce composite web services, whereas orchestration produces a workflow from web services. Of course, the resulting workflow can be made into another web service, but this has to be done on top of orchestration (Lau and Tran, 2008). Additionally, we can build systems step by step and the generated subsystems are logically highly cohesive, low coupled and documented. Thus, during the development process, a change can be localised to a subsystem rather than globalised to the entire workflow. In our case it only requires the modification of the connector definition, as components do not contain connectors' code.

8.3. Desirable characteristics of composition and adaptation mechanisms

For evaluation purposes, it is also interesting to discuss whether the connectors in the catalogue have the desirable characteristics for composition and adaptation mechanisms. In Chaudron (2001) a list of six recommendations for the design of software composition languages and mechanisms is presented. Although the recommendations are not exclusively for components, we consider they are worthy of consideration. The recommendations are:

(1) composition should be exogenous to components,
(2) composition should be non-intrusive,
(3) composition mechanisms should provide separate mechanisms for dealing with data flow and control flow,
(4) composition languages should provide means for building higher-level larger granularity composition abstractions,
(5) composition mechanisms should provide support for bridging multiple control and data flow styles and
(6) composition mechanisms are subject to general quality requirements such as timeliness, reliability, extensibility, scalability etc.

Table 8 summarises how many of these recommendations are followed by current composition approaches based on first-class connectors, including ours. As can be seen, exogenous connectors follow most of them. Recommendations (1), (2) and (3) are strongly supported by exogenous mechanisms for dealing with data flow and control flow. We have not yet implemented recommendations (5) and (6); however, by defining connectors that encapsulate communication and coordination schemes to support concurrency and parallelism as well as defining non-functional properties for the proposed connectors, they can be achieved.

Note that, we are the only supporters of recommendations (2) and (4). In contrast to other approaches, our connectors are indeed exogenous to components during all the development stages. Thus, we do not need to insert their code into components, as recommendation (2) states. Our approach for constructing composite connectors meets recommendation (4). In C2 two bus connectors can be connected. However, it does not result in a new connector type as in our approach. In SOFA a connector is composed from a set of *connector elements*. The elements model non-functional properties of some basic connector types supported in middleware technologies. In the work by Spitznagel, an ADL connector can be adapted by composing a set of transformations. The transformations can modify the connector's properties, e.g. protocol, data policy. Both SOFA and Spitznagel's work deal only with the construction of a single connector rather than the construction of

Table 8
Recommendations for the design of software composition languages and mechanisms according to Chaudron (2001).

	(1)	(2)	(3)	(4)	(5)	(6)
UniCon	√	×	√	×	×	√
C2	√	×	√	×	×	√
CLARA	√	×	√	×	×	√
CoCo	√	×	√	×	×	×
SOFA	√	×	√	×	×	√
Spitznagel	√	×	√	×	√	√
Schreiner and Goschka	√	×	√	×	×	
Wydaeghe and Vanderperren	√	×	√	×	×	×
Our Composition Connectors	√	√	√	√	√[a]	√[b]

[a] Feasible via connectors encapsulating control schemes to support parallelism and concurrency.

[b] Feasible by extending connectors' properties to include non-functional ones.

a composite. The issue of connector composition is not tackled in the rest of the approaches.

Similarly, (Heineman and Ohlenbusch, 1999) discussed eleven requirements for adaptation techniques:

(1) *Homogeneous*: the code that uses an adapted component should use it in the same manner as it would have used the non-adapted version of it.
(2) *Conservative*: the aspects of a component that were not adapted should be accessible without explicit effort by the adapted version of it.
(3) *Ignorant*: a component should have no knowledge of its adaptations.
(4) *Identity*: a component should continue to retain its own identity as a separate entity; this eases the way in which future updates of the component will be handled.
(5) *Composable*: a component should be open to future adaptations; it should be straightforward to compose together a set of desired adaptations.
(6) *Configurable*: the adaptation technique should be able to parametrise and apply a particular adaptation to many different components.
(7) *Black-box*: the adaptation technique should have no knowledge of the internal implementation of a component.
(8) *Architectural focus*: there should be a global description of the architecture of the target application together with different specifications of both the non-adapted and the adapted version of a component.
(9) *Framework independent*: the adaptation technique must not be dependent upon the component framework to which the component belongs.
(10) *Embedded*: the adaptation mechanism must exist within the component before it can be adapted.
(11) *Language independent*: the adaptation mechanism must not be dependent upon the language used to implement it.

Table 9 summarises how many of these requirements are met by our adaptation connectors. In contrast to Table 8, other approaches are not included in the table because they do not use connectors as adaptation mechanisms. As can be seen, most of the requirements are fulfilled by our adaptation connectors. The requirement (2) cannot be satisfied because an adaptation connector adapts the whole component. The requirement (10) goes against the exogenous nature of our adaptation connectors.

8.4. Scope of connectors in the catalogue

In all the above mentioned work on connectors, the main concern is "composition"; they do not focus on adaptation or any other issue relevant to the life cycle of CBD. Thus, we consider that our catalogue is unique because it has a bigger scope. That is, it includes a set of reusable connector types to support not only *composition* but also *adaptation* of components. We believe that it represents a step forward because in all related work, there is no similar catalogue.

Besides that, we can say that our connectors catalogue is *turing complete* (Le Metayer et al., 1998; Böhm and Jacopini, 1966) because it includes the three standard control structures in programming languages: *sequencing*, *branching* and *looping*. Thus, by using our catalogue a wide variety of systems can be developed. This claim is supported by Velasco Elizondo (2009) and Lau and Wang (2007) where our connectors have been utilised to develop robotics and a missile guidance systems, respectively. Note that all our composite connectors are constructed from these three basic control structures. The exact number and nature of connectors in the catalogue determine only their usefulness for particular applications. More connectors can always be defined and added to the catalogue as and when appropriate or needed.

A limitation of the connectors in our catalogue is that they are *general purpose* connectors defined within the context of a *sequential* and *single thread model*. That is, neither domain specificity nor concurrency issues have been considered in this work. It impacts on the variety of systems that can be constructed by using the proposed catalogue. Additionally, some connectors impose syntactic constraints on the components they compose, e.g. the data communication characteristic of the *pipe* connector requires the output generated by a component's execution and the input to the next one in the chain to have the same type. Thus, a set of components cannot be composed by a Pipe if they do not fulfil this syntactic constraint. We are convinced we need to consider a solution that overcomes this limitation, e.g. the use of data connectors for transforming data.

Despite these limitations, we are convinced that our connectors are useful for developing certain types of control systems. For example, real time control systems with an underlying closed control loop architecture (Shaw, 1995). In general, a closed control-loop architecture includes three main elements: a controller component – which continually receives information about the physical system and supplies continuous guidance about the changes to be performed to maintain its properties; a sensor component – which is engaged in gathering information about the physical system and sending it to the Controller; and an actuator component – which is involved in performing the decisions taken by the controller. All these three elements are connected by connectors that pass on the corresponding data, usually modelled as primitive data types. By composing components like ours, which exhibit very simple interfaces that solely receive and provide primitive data types, via our connectors we can produce an alternative architecture to model the behaviour of the closed control loop.

Consider the security control system discussed in Section 7.4. We do not have a controller component in our architecture as the connectors encapsulate all the decision making. However, we maintain the simplicity of the style by keeping the sensor (the subsystem responsible for sensing the status of all the detectors MD, IRD and SD) and the actuators (the subsystem responsible for firing the alarm A and the sprinkler S in the event of an emergency) separated from each other, which facilitates their independent treatment. Note that all component interactions in the security control system are interface-level and are determined by the semantics of the new component model. Within this semantics, the actual information flow between interfaces is represented by individual variables and primitive data types as illustrated in Fig. 9 and Program 1.

Additionally, our connectors provide a means to refine the steps of sensing, decision making and acting if necessary. Note that we have done it using the *simple merge sequencer* connector SMS to control the execution of the computation in only one of the motion

Table 9
Requirements for adaptation techniques according to Heineman and Ohlenbusch (1999).

	(1)	(2)	(3)	(4)	(5)	(6)	(7)	(8)	(9)	(10)	(11)
Adaptation connectors	√	×	√	√	√	√	√	√	√	×	√

detector components depending on the operation mode of the system. The inability of performing such refinements has been recognised as a shortcoming in closed control loop architectures (Shaw, 1995).

8.5. Characteristics of our development approach and the generated systems

With our connectors, system construction is a pure *bottom-up* process involving not only pre-existing components, but also pre-existing connectors. This requires a new approach to implementing components that is completely different from current component-based development approaches. This new way of thinking may limit the immediate adoption of our approach in practice. However, current experiences in CBD demonstrate that the manner in which components and connectors are treated in existing component models is not enough to achieve the CBD desiderata, i.e. components should be pre-existing reusable units of functionality, be produced and used by independent parties, be composable into composite components and be distinguished from their instances (Meyer, 2003; Szyperski et al., 2002; Heineman and Councill, 2001; Broy et al., 1998). That makes the proposed approach worthy of consideration.

Another positive characteristic of our connectors is that they allow the definition and generation of subsystems that are logically highly cohesive but low-coupled at the same time. These subsystems are compositional and can also be well documented. Any generated subsystem is an identifiable compilation unit that offers certain functionality which is described by its interface. As described in Section 7.4, these interfaces are generated according to the semantics of the connector utilised and documented by adding the corresponding annotations into the resulting (sub)systems' code. Therefore, the resulting subsystems can be tested and (if required) reused separately for further composition or adaptation. This has been illustrated by the security system described in Section 7.4. Many of the related approaches to composition fail to tackle these issues because they do not have connectors that work as composition operators. This makes it necessary for them to adopt alternative methods to generate the subsystem interfaces separately in order to allow the individual treatment of the generated assemblies.

Similarly, the issue of static evolution of system architectures can be better dealt with in our approach. Due to the fact that all component and connector interactions are interface-level, changing a connector in a system hierarchy has consequences for its adjacent architectural elements only when such interface-level rules are not fulfilled. Even then, these changes can be dealt with within the scope of a subsystem, and it only requires the modification of the connector definition in the subsystem because components do no contain connectors' code. As a corollary, these changes can be tested separately.

An obvious disadvantage of the connectors in our catalogue is the potential preponderance of connectors and connector levels, and hence inefficiency in communication. Some systems could require a complex hierarchy which might have a negative impact on the runtime performance. However, our connectors allow and promote compositionality and separation of concerns, which are the key desiderata of CBD.

9. Conclusion and future work

In this paper we have presented a catalogue of connectors to support the process of development with reuse within the context of CBD. The catalogue includes a set of thirteen connector types that are defined by taking into consideration the syntax and semantics of a new component model. The feasibility of implementing the catalogue and its use in practice has been positively demonstrated.

We consider that our catalogue possesses several good characteristics that make it unique. Firstly, the connectors in our catalogue are indeed composition operators for components; that is, when applied to components they yield another component. Secondly, our catalogue has a bigger scope than existing ones because it includes a set of connector types to support not only composition but also adaptation of components. Thirdly, all the connectors in our catalogue are reusable not only at design but also at implementation stage. Fourthly, our connectors meet most of the requirements and recommendations in the literature for both composition and adaptation mechanisms. Finally, and as a corollary of all the above, our catalogue improves on current CBD practice because our connectors allow a bottom-up approach to system construction from pre-existing reusable components. It is important to highlight that such a bottom-up approach involves not only pre-existing components, but also pre-existing connectors. Thus, reuse is maximised in the process of development with reuse.

Our work contributes new knowledge to the area of CBD and Software Engineering not only by defining a novel approach for developing component-based systems but also by defining a new view into the space of software connectors where they are indeed first-class entities during all development stages.

For future work, we plan to consider the issues of domain specificity (Lau and Taweel, 2009) and non-functional properties for components and connector definitions. In Section 8.4, we mentioned that we are convinced that our connectors are good for developing certain types of control systems. In all our previous work neither the component nor the connector specifications include information about non-functional issues, which will make them completely suitable for control systems construction. We have already started some work on augmenting the information in a component interface to further specify a service provided in terms of the *worst-case execution time* considering the *CPU* as the main context dependency and integrating the proper analysis engines in our development tool.

Also, we plan to consider other communication and coordination schemes, e.g. concurrency, which will lead to new and perhaps domain specific connector types.

Finally, we are also considering the issue of producing a kind of "control components" that emulate the connectors by using some other component model, e.g. EJB. This exercise will help to evaluate the feasibility of implementing the semantics of our component by using the infrastructure provided by some existing component model and as a corollary, it could help to motivate people to use our component model.

References

Atkinson, C., Bayer, J., Bunse, C., Kamsties, E., Laitenberger, O., Laqua, R., Muthig, D., Paech, B., Wust, J., Zettel, J., 2001. Component-Based Product Line Engineering with UML. Addison-Wesley.

Bass, L., Clements, P., Kazman, R., 2003. Software Architecture in Practice, second ed. Addison-Wesley.

Becker, S., Overhage, S., Ralf, R., 2004. Classifying software component interoperability errors to support component adaption. In: Crnkovic, I., Stafford, J., Schmidt, H., Wallnau, K. (Eds.), Proceedings of the Component-Based Software Engineering, Seventh International Symposium (CBSE), Springer, pp. 68–83.

Becker, S., Brogi, A., Gorton, I., Overhage, S., Romanovsky, A., Tivoli, M., 2006. Towards an engineering approach to component adaptation. Technical Report CS-TR-939, University of Newcastle upon Tyne, Computing Science.

Böhm, C., Jacopini, G., 1966. Flow diagrams, turing machines and languages with only two formation rules. Communications of the ACM 9 (5), 366–371.

Bosch, J., 1999. Superimposition: a component adaptation technique. Information and Software Technology 41 (5), 257–273.

Broy, M., Deimel, A., Henn, J., Koskimies, K., Plasil, F., Pomberger, G., Pree, W., Stal, M., Szyperski, C., 1998. What characterizes a (software) component? Software – Concepts and Tools 19 (1), 49–56.

Bures, T., 2006. Generating connectors for homogeneous and heterogeneous deployment. Ph.D. Thesis, Department of Software Engineering, Faculty of Mathematics and Physics, Charles University in Prague, Prague, Czech Republic.

Chaudron, M., 2001. Reflections on the anatomy of software composition languages and mechanism. In: Proceedings of the Workshop on Composition Languages, pp. 45–53.

Clements, P., 1996. A survey of architecture description languages. In: Proceedings of the Eighth International Workshop on Software Specification and Design (IWSSD), IEEE Computer Society, pp. 16–25.

CORBA Component Model Web Page, <http://www.omg.org/technology/documents/formal/components.htm>.

Durand, E., Déplanche, A., 1999. CLARA – An architecture description language for real-time applications. Technical Report, IRCCyN.

EJB Web Page, <http://java.sun.com/products/ejb/>.

Gamma, E., Helm, R., Johnson, R., Vlissides, J., 1995. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley Professional Computing Series.

Gast, H., 2008. Patterns and traceability in teaching software architecture. In: Proceedings of the Sixth International Symposium on Principles and Practice of Programming in Java (PPPJ), ACM, New York, NY, USA, pp. 23–31.

Heineman, G.T., 2000. A model for designing adaptable software components. In: Proceedings of the 22nd Annual International Computer Software and Applications Conference, ACM Press, New York, NY, USA, pp. 55–56.

Heineman, G.T., Councill, W.T. (Eds.), 2001. Component-Based Software Engineering: Putting the Pieces Together. Addison-Wesley.

Heineman, G., Ohlenbusch, H. (Eds.), 1999. An evaluation of component adaptation techniques. Technical Report WPI-CS-TR-98-20, Department of Computer Science, Worcester Polytechnic Institute.

Kim, J.A., Lee, K.W., 2005. Binary component adaptation technique and supporting tool. In: Proceedings of the Third ACIS International Conference on Software Engineering Research, Management and Applications (SERA), IEEE Computer Society, Washington, DC, USA, pp. 14–21.

Kuchana, Partha, 2004. Software Architecture Design Patterns in Java. Auerbach Publications, Boston, MA, USA.

Lau, K.-K., Ornaghi, M., 2009. Control encapsulation: a calculus for exogenous composition. In: Lewis, G., Poernomo, I., Hofmeister, C. (Eds.), Proceedings 12th International Symposium on Component-based Software Engineering, LNCS 5582, Springer-Verlag, pp. 121–139.

Lau, K.-K., Taweel, F.M., 2009. Domain-specific software component models. In: Lewis, G., Poernomo, I., Hofmeister, C. (Eds.), Proceedings of the 12th International Symposium on Component-based Software Engineering, LNCS 5582, Springer-Verlag, pp. 19–35.

Lau, K.-K., Tran, C.M., 2008. Composite web services. In: Pautasso, C., Gschwind, T. (Eds.), Emerging Web Services Technology, vol. 2. Birkhauser Verlag, pp. 77–95.

Lau, K.-K., Wang, Z., 2007. Software component models. IEEE Transactions on Software Engineering 33 (10), 709–724.

Lau, K.-K., Wang, Z., 2007. Verified component-based software in SPARK: experimental results for a missile guidance system. In: Proceedings of the 2007 ACM SIGAda Annual International Conference, ACM, pp. 51–57.

Lau, K.-K., Velasco Elizondo, P., Wang, Z., 2005. Exogenous connectors for software components. In: Heineman, G.T., Crnkovic, I., Schmidt, H., Stafford, J., Szyperski, C., Wallnau, K. (Eds.), Proceedings of Eighth International SIGSOFT Symposium on Component-based Software Engineering, Springer-Verlag Heidelberg, pp. 90–106.

Lau, K.-K., Ornaghi, M., Wang, Z., 2006. A software component model and its preliminary formalisation. In: de Boer, F.S., et al. (Eds.), Proceedings of the Fourth International Symposium on Formal Methods for Components and Objects, vol. 4111 of Lecture Notes in Computer Science. Springer-Verlag, pp. 1–21.

Lau, K.-K., Ling, L., Velasco Elizondo, P., 2007. Towards composing software components in both design and deployment phases. In: Schmidt, H.W., et al. (Eds.), Proceedings of the 10th International Symposium on Component-based Software Engineering, LNCS 4608, Springer, pp. 274–282.

Lau, K.-K., Ling, L., Velasco Elizondo, P., Ukis, V., 2007. Composite connectors for composing software components. In: Lumpe, M., Vanderperren, W. (Ed.), Proceedings of the Sixth International Symposium on Software Composition, LNCS 4829, Springer-Verlag, pp. 266–280.

Le Metayer, D., Nicolas, V.-A., Ridoux, O., 1998. Programs, properties, and data: exploring the software development trilogy. IEEE Software 15 (6), 75–81.

Medvidovic, N., Taylor, R.N., 2000. A classification and comparison framework for software architecture description languages. Software Engineering 26 (1), 70–93.

Meyer, B., 2003. The grand challenge of trusted components. In: Proceedings of the 25th International Conference on Software Engineering, IEEE Computer Press, pp. 660–667.

Monroe, R., Garlan, D., Wile, D., 1997. Acme: an architecture description interchange language. In: Proceedings of CASCON'97, Toronto, Ontario, pp. 169–183.

Murta, L.G.P., van der Hoek, A., Werner, C.M.L., 2006. ArchTrace: policy-based support for managing evolving architecture-to-implementation traceability links. In: Proceedings of the 21st IEEE/ACM International Conference on Automated Software Engineering (ASE), IEEE Computer Society, Washington, DC, USA, pp. 135–144.

Ommering, R.V., Linden, F.V., Kramer, J., Magee, J., 2000. The Koala component model for consumer electronics software. IEEE Computer, 78–85.

Nierstrasz, O., Arevalo, G., Ducasse, S., Wuyts, R., Black, A., Muller, P., Zeidler, C., Genssler, T., van den Born R., 2002. A component model for field devices. In: Proceedings of the First International IFIP/ACM Working Conference on Component Deployment.

Richardson, J., Green, J., 2004. Automating traceability for generated software artifacts. In: Proceedings of the 19th IEEE International Conference on Automated Software Engineering (ASE), IEEE Computer Society, Washington, DC, USA, pp. 24–33.

Russell, N., ter Hofstede, A.H.M., van der Aalst, W.M.P., Mulyar, N., 2006. Workflow control-flow patterns: A revised view. Technical Report 34, BPM Center, BPM-06-22.

UniCon Web Page, <http://www-2.cs.cmu.edu/afs/cs/project/vit/www/unicon/>.

Schreiner, D., Göschka, K.M., 2007. Explicit connectors in component-based software engineering for distributed embedded systems. In: Proceedings of the 33rd International Conference on Current Trends in Theory and Practice of Computer Science (SOFSEM), volume 2436 of Lecture Notes in Computer Science, Springer-Verlag Heidelberg, pp. 923–934.

Shaw, M., 1995. Beyond objects: a software design paradigm based on process control. ACM SIGSOFT Software Engineering Notes 20 (1), 27–38.

Shaw, M., Garlan, D., 1996. Software Architecture: Perspectives on an Emerging Discipline. Prentice-Hall.

Sommerville, I., 2004. Software Engineering, seventh ed. Pearson Addison Wesley.

Spitznagel, B., 2004. Compositional Transformation of Software Connectors. Ph.D. Thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, USA.

Streitferdt, D., 2001. Traceability for system families. In: ICSE '01: Proceedings of the 23rd International Conference on Software Engineering (ICSE), IEEE Computer Societym Washington, DC, USA, pp. 803–804.

JavaBeans Web Page, <http://java.sun.com/products/javabeans/>.

Java TM 2 Platform Standard Ed. 5.0., %3chttp://java.sun.com/j2se/1.5.0/docs/api/java/lang/reflect/package-summary.html%3e.

Szyperski, C., Gruntz, D., Murer, S., 2002. Component Software: Beyond Object-Oriented Programming, second ed. Addison-Wesley.

Tansalarak, N., Claypool, K., 2005. CoCo: composition model and composition model implementation. In: Proceedings of the Seventh International Conference on Enterprise Information Systems, pp. 340–345.

The C2 Style Web Page, <http://www.isr.uci.edu/architecture/c2.html>.

Velasco Elizondo, P., 2008. Component composition in the deployment phase with exogenous connectors. Ph.D. Thesis, School of Computer Science, The University of Manchester, Manchester, UK.

Velasco Elizondo, P., 2009. Systematic and automated development with reuse, <http://www.cimat.mx/pvelasco/exo/exotool_en.html>.

Wydaeghe, B., Vanderperren, W., 2001. Visual component composition using composition patterns. In: Proceedings of the 39th International Conference and Exhibition on Technology of Object-Oriented Languages and Systems (TOOLS), IEEE Computer Society, pp. 120–129.

Kung-Kiu Lau holds a PhD degree from the University of Leeds, UK. After a temporary appointment at Leeds, he moved to the University of Manchester, UK, where he is a senior lecturer. He is the series editor of a book series on Component-based Software Development and an area editor of the Journal of Applied Logic. He has served on numerous programme committees (e.g. the International Automated Software Engineering Conference 2000–2005, the International Symposium on Component-based Software Engineering 2005–2009, the International Symposium on Software Composition 2006–2007, and the International Conference on Generative Programming and Component Engineering 2009). Similarly, he has delivered invited talks and tutorials at many international meetings (e.g. Twentieth IEEE/ACM International Conference on Automated Software Engineering 2005 and the Twenty-eighth International Conference on Software Engineering 2006). He was programme chair for the Seventh International Conference on Formal Engineering Methods 2005, and for the Component-based Software Engineering Track at the Thirty-fourth Euromicro Conference on Software Engineering and Advanced Applications 2008. His main research areas are Component-based Software Development and Formal Program Development in Computational Logic.

Perla Velasco Elizondo holds a PhD degree from the University of Manchester, UK. Since 2008, she is a full-time researcher at CIMAT as well as faculty member of the Master in Software Engineering program at this institution. She is member of the National System of Researchers in Mexico (candidate level). From 2004-2007, she served as external reviewer for various international conferences (e.g. CBSE 2007, SC 2007, ASE 2004). Most recently she served as reviewer for the Journal of System and Software (2009) and the Jounal of Zhejian University SCIENCE (2007). Her research interest include Component-based Software Engineering, Architecture-Centered Software Development, Software Reuse, Automated Software Development and Software Engineering Education.