A Software Component Model and Its Preliminary Formalisation

Kung-Kiu Lau[1], Mario Ornaghi[2], and Zheng Wang[1]

[1] School of Computer Science, the University of Manchester
Manchester M13 9PL, United Kingdom
{kung-kiu, zw}@cs.man.ac.uk
[2] Dipartimento di Scienze dell'Informazione,
Universita' degli studi di Milano
Via Comelico 39/41, 20135 Milano, Italy
ornaghi@dsi.unimi.it

Abstract. A software component model should define what components are, and how they can be composed. That is, it should define a theory of components and their composition. Current software component models tend to use objects or port-connector type architectural units as components, with method calls and port-to-port connections as composition mechanisms. However, these models do not provide a proper composition theory, in particular for key underlying concepts such as encapsulation and compositionality. In this paper, we outline our notion of these concepts, and give a preliminary formalisation of a software component model that embodies these concepts.

1 Introduction

The context of this work is Component-based Software Engineering, rather than Component-based Systems. In the latter, the focus is on system properties, and components are typically state machines. Key concerns are issues related to communication, concurrency, processes, protocols, etc. Properties of interest are temporal, non-functional properties such as deadlock-freedom, safety, liveness, etc. In the former, the focus is on software components and middleware for composing them. Usually a software component model, e.g. Enterprise JavaBeans (EJB) [21], provides the underlying framework.

A software component model should define (i) what components are, i.e. their syntax and semantics; and (ii) how to compose components, i.e. the semantics of their composition. Current component models tend to use objects or port-connector type architectural units as components, with method calls and port-to-port connections as composition mechanisms. However, these models do not define a proper theory for composition.

We believe that encapsulation and compositionality are key concepts for such a theory. In this paper, we explain these notions, and their role in a composition theory. Using these concepts, we present a software component model, together with a preliminary formalisation.

2 Current Component Models

Currently, so-called component models, e.g. EJB and CCM (CORBA Component Model) [24], do not follow a standard terminology or semantics. There are different definitions of what a component is [6], and most of these are not set in the context of a component model. In particular, they do not define composition properly.

For example, a widely used definition of components is the following, due to Szyperski [28]:

> "A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties."

A different definition is the following by Meyer [20]:

> "A component is a software element (modular unit) satisfying the following conditions:
> 1. It can be used by other software elements, its 'clients'.
> 2. It possesses an official usage description, which is sufficient for a client author to use it.
> 3. It is not tied to any fixed set of clients."

Both these definitions do not mention a component model, in particular how composition is defined.

The following definition given in Heineman and Councill [12] mentions a component model:

> "A [component is a] software element that conforms to a component model and can be independently deployed and composed without modification according to a composition standard."

but it does not define one.

Nevertheless, there is a commonly accepted abstract view of what a component is, viz. a software unit that contains (i) code for performing services, and (ii) an interface for accessing these services (Fig. 1(a)). To provide its services, a component

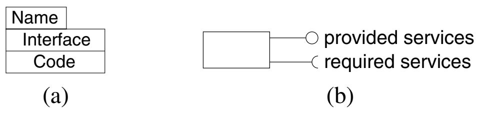


Fig. 1. A software component

may require some services. So a component is often depicted as in Fig. 1(b), e.g. in CCM and UML2.0 [23].

In current software component models, components are typically objects as in object-oriented languages, and port-connector type architectural units, with method calls and ADL (architecture description languages [26]) connectors as composition mechanisms respectively.

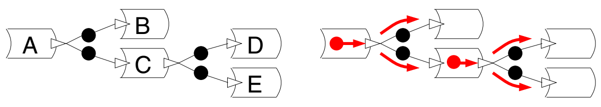
A complete survey of these models is beyond the scope of this paper. It can be found in [17].

3 Our Component Model

In our component model, *components* encapsulate *computation* (and data),[1] and *composition operators* encapsulate *control*. Our components are constructed from (i) computation units and (ii) connectors. A computation unit performs computation within itself, and does not invoke computation in another unit. Connectors are used to build components from computation units, and also as composition operators to compose components into composite components.

3.1 Exogenous Connectors

Our connectors are *exogenous connectors* [16]. The distinguishing characteristic of exogenous connectors is that they encapsulate control. In traditional ADLs, components



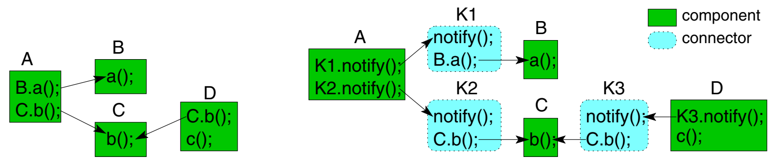
(a) Components and connectors (b) Control flow

Fig. 2. Traditional ADLs

are supposed to represent *computation*, and connectors *interaction* between components [19] (Fig. 2 (a)). Actually, however, components represent computation as well as *control*, since control originates in components, and is passed on by connectors to other components. This is illustrated by Fig. 2 (b), where the origin of control is denoted by a dot in a component, and the flow of control is denoted by arrows emanating from the dot and arrows following connectors.

In this situation, components are not truly independent, i.e. they are tightly coupled, albeit only indirectly via their ports.

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



(a) Direct message passing (b) Indirect message passing

Fig. 3. Connection by message passing

passing corresponds to direct method calls, as exemplified by objects calling methods in other objects (Fig. 3 (a)), using method or event delegation, or remote procedure call (RPC). Software component models that adopt direct message passing schemes as

[1] For lack of space, we will not discuss data in this paper.

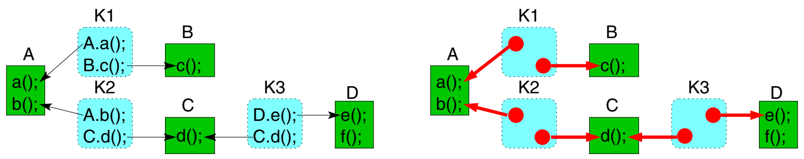
composition operators are EJB, CCM, COM [5], UML2.0 [23] and KobrA [3]. In these models, there is no explicit code for connectors, since messages are ‘hard-wired’ into the components, and so connectors are not separate entities.

Indirect message passing corresponds to coordination (e.g. RPC) via connectors, as exemplified by ADLs. Here, connectors are separate entities that are defined explicitly. Typically they are glue code or scripts that pass messages between components indirectly. To connect a component to another component we use a connector that when notified by the former invokes a method in the latter (Fig. 3 (b)). Besides ADLs, other software component models that adopt indirect message passing schemes are JavaBeans [27], Koala [30], SOFA [25], PECOS [22], PIN [14] and Fractal [8].

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

A categorical semantics of connectors is proposed in [9], where coordination is modelled through signature morphisms. There is a clear separation between computation, occurring in components, and coordination, performed by connectors. However, shared actions may propagate control from one component to others.

By contrast, in exogenous connection, control originates in and flows from connectors, leaving components to encapsulate only computation. This is illustrated by Fig. 4.



(a) Example (b) Control flow

Fig. 4. Connection by exogenous connectors

In Fig. 4 (a), components do not call methods in other components. Instead, all method calls are initiated and coordinated by exogenous connectors. The latter’s distinguishing feature of control encapsulation is clearly illustrated by Fig. 4 (b), in clear contrast to Fig. 2 (b).

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

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

3.2 Components

Our view of a component is that it is not simply a part of the whole system. Rather it is something very different from traditional software units such as code fragments, functions, procedures, subroutines, modules, classes/objects, programs, packages, etc, and equally different from more modern units like DLLs and services.

We define a component as follows:

Definition 1. A *software component* is a software unit with the following defining characteristics: (i) encapsulation and (ii) compositionality.

A component should encapsulate both *data* and *computation*. A component C encapsulates data by making its data private. C encapsulates computation by making sure that its computation happens entirely within itself.

An object can encapsulate data, but it does not encapsulate computation, since objects can call methods in other objects (Fig. 5(a)).

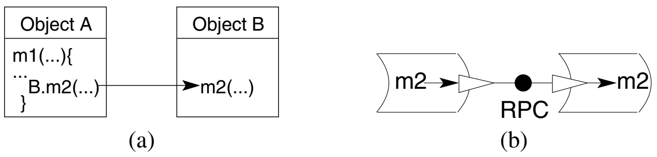


Fig. 5. Objects and architectural units

Port-connector type components, as in e.g. ADLs, UML2.0 and Koala, can encapsulate data. However, they usually do not encapsulate computation, since components can call methods in other components by remote procedure call (RPC), albeit only indirectly via connectors (and ports) (Fig. 5(b)).

Components should be *compositional*, i.e. the composition of two components C_1 and C_2 should yield another component C_3, which in turn should also have the defining characteristics of encapsulation and compositionality. Thus compositionality implies that composition preserves or propagates encapsulation.[2]

Classes and objects are not compositional. They can only be ‘composed’ by method calls, and such a ‘composition’ does not yield another class or object. Indeed, method calls break encapsulation. Port-connector type components can be composed, but they are not compositional if they do not have (computation) encapsulation.

Encapsulation entails that access to components must be provided by *interfaces*. Classes and objects do not have interfaces. Access to (the methods of) objects, if permitted, is direct, not via interfaces. So-called ‘interfaces’ in object-oriented languages like Java are themselves classes or objects, so are not interfaces to components. Port-connector type components use their ports as their interfaces.

Our components are constructed from *computation units* and *exogenous connectors*. A computation unit performs just computation within itself and does not invoke computation in another unit. It can be thought of as a class or object with methods, except that these methods do not call methods in other units. Thus it encapsulates computation.

Exogenous connectors encapsulate control, as we have seen in the previous section. The type hierarchy of these connectors in our component model is as follows. At the

[2] Compositionality in terms of other (non-functional) properties of sub-components is an open issue, which we do not address here.

lowest level, level 1, because components are not allowed to call methods in other components, we need an exogenous *invocation connector*. This is a *unary* operator that takes a computation unit, invokes one of its methods, and receives the result of the invocation. At the next level of the type hierarchy, to structure the control and data flow in a set of components or a system, we need other connectors for sequencing exogenous method calls to different components. So at level 2, we need n-*ary* connectors for connecting invocation connectors, and at level 3, we need n-*ary* connectors for connecting these connectors, and so on. In other words, we need a hierarchy of connectors of different arities and types. We have defined and implemented such a hierarchy in [16]. Apart from invocation connectors at level 1, our hierarchy includes *pipe* connectors, for sequencing, and *selector* connectors, for branching, at levels $n \geq 2$. These connectors are called *composition connectors* for the obvious reason. Level-1 connectors are invocation connectors, and level-2 composition connectors connect only invocation connectors, but composition connectors at higher levels are polymorphic since they can connect different kinds of connectors at different levels (and with different arities).

We distinguish between (i) *atomic* components and (ii) *composite* components.

Definition 2. An *atomic component* C is a pair $\langle i, u \rangle$ where u is a computation unit, and i is an invocation connector that invokes u's methods. i provides an interface to the component C.

A *composite component* CC is a tuple $\langle k, C_1, C_2, \ldots C_j \rangle$, for some j, where k is a j-ary connector at level $n \geq 2$, and each $C_i, i = 1, \ldots, j$, is either an atomic component or a composite component. k is called a *composition connector*. It provides an interface to the component CC.

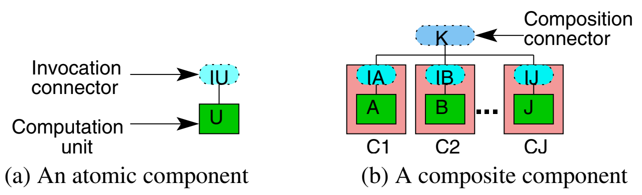


Fig. 6. Atomic and composite components

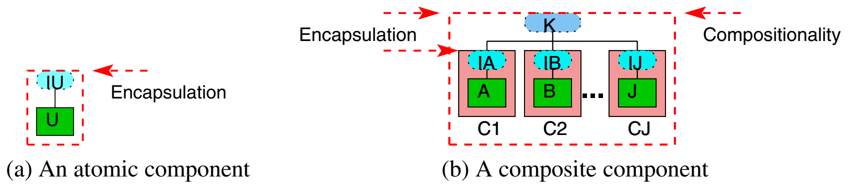


Fig. 7. Encapsulation and compositionality

Figure 6 illustrates atomic and composite components. Clearly, an atomic component encapsulates computation, since a computation unit does so, and an invocation connector invokes only methods in the unit (Fig 7(a)). It is easy to see that a composite component also encapsulates computation (Fig 7(b)).

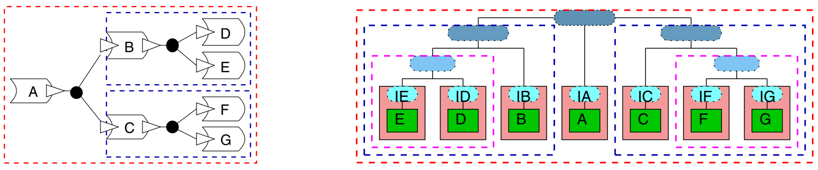
3.3 Composition Operators

To construct systems or composite components, we need composition operators that preserve encapsulation and compositionality. Such composition operators should work only on interfaces, in view of encapsulation.

Glue code is certainly not suitable as composition operators. Neither are object method calls or ADL connectors, as used in current component models. Indeed, these models do not have proper composition operators, in our view, since they do not have the concepts of encapsulation and compositionality.

As in Definition 2, we use exogenous connectors at level $n \geq 2$ as composition operators. These operators are compositional and therefor preserve and propagate encapsulation. As shown in Fig 7(b), a composite component has encapsulation, as a result of encapsulation in its constituent components. Furthermore, the composite component is also compositional. Thus, any component, be it atomic or composite, has a top-most connector that provides an interface, and it can be composed with any other component using a suitable composition operator.

This self-similarity of a composite component is a direct consequence of component encapsulation and compositionality, and provides the basis for a compositional way of constructing systems from components. Fig. 8(b) illustrates self-similarity of a composite component in a system composed from atomic and composite components. Each dotted box indicates a composite component. Note in particular that the composite at the top level is the entire system itself. Most importantly, every composite component is similar to all its sub-components.



(a) Acme (b) Exogenous connection

Fig. 8. Self-similarity of a composite component

The system in Fig. 8(b) corresponds to the Acme [11] architecture in Fig. 8(a). By comparison, in the Acme system, the composites are different from those in Fig. 8(b). For instance, (D,E) is a composite in (b) but not in (a). Also, in (a) the top-level composite is not similar to the composite (B,D,E) at the next level down. The latter has an interface, but the former does not.

In general, self-similarity provides a compositional approach to system construction, and this is an advance over hierarchical approaches such as ADLs which are not compositional, strictly speaking.

3.4 The Bank Example

Having defined our component model, we illustrate its use in the construction of a simple bank system. The bank system has just one ATM that serves two banks (Bank1 and Bank2) as shown in Fig. 9.

The ATM obtains client details including card number, PIN, and requested services, etc., and locates the bank that the client account belongs to. It then passes client details to the client's bank, which then provides the requested services of withdrawal, deposit, etc.

To construct the bank system, first, two Bank components, Bank1 and Bank2, are assembled by a Selector connector into a BankComposite. Then BankComposite is composed with an ATM component by a Pipe connector into the bank system.

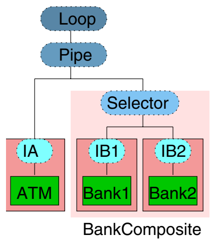


Fig. 9. A simple bank system

All the components in Fig. 9 are well encapsulated. Every atomic component is made up of an invocation connector and a computation unit. The computation unit implements methods that the component could offer, and the invocation connector provides functionalities to invoke those methods of the computation unit; thus it provides the interface to the component. For example, the ATM component is made up of a computation unit ATM and an invocation connector IA. The ATM component encapsulates both data and functions, by defining its data as private and computing its functions entirely within itself. IA invokes methods of ATM and thus serves as its interface.

All the components in Fig. 9 are compositional. The composite component BankComposite is itself a component, with the composition connector Selector as its interface. BankComposite also has encapsulation: it encapsulates data and functions of its constituent components, Bank1 and Bank2.

Moreover, the composite component BankComposite in Fig. 9 is self-similar. BankComposite has a top-level connector as an interface, and so have its sub-components Bank1 and Bank2.

Finally, in this example, we built the bank system with a loop connector at the outermost level, which iterates and waits for client requests.

4 A Preliminary Formalisation

So far we have defined our component model informally. In this section we present a preliminary formalisation of our model. The formalisation serves as a useful check on the soundness, in the sense of good judgement, of the underlying ideas.

We will assume that our component model provides the basis for a Component-based Software Design System, which we will call DS for convenience. DS should support all the phases of the component life-cycle [18], i.e.:

(i) *Design Phase*. A system or a new component C is designed from the connectors, the computation units, and/or existing components, in a *repository*. If C is a new component, it is added to the repository.
(ii) *Deployment Phase*. Deployment prepares a system or a component C for execution, i.e. turns it into a *binary* b_C and establishes the way of *loading* and instantiating it when it is launched.
(iii) *Run-time Phase*. A binary b_C is launched by loading and instantiating it into a running instance, which provides the services offered by C.

Deployment-phase compositionality should be supported by suitable deployment tools and should follow from design-phase compositionality. For this, a DS should be provided with a composition and run-time infrastructure, implementing deployment-phase composition according to the design-phase semantics of connectors.

Here we outline a first abstract meta-model for the design phase, and discuss functional compositionality and encapsulation. The meta-model is formalised in many-sorted first-order logic with identity. In the explanation we will introduce the signature incrementally and we will explain the intended meaning informally. Axioms are given in the appendix. The purpose is to devise and establish the basic requirements to be satisfied by DS and the design-phase semantics of connectors.

4.1 Components and Their Interfaces

DS should provide a design environment for components with the characteristics of compositionality and encapsulation, as discussed in Section 3.2. Component interfaces play a key role, for two main reasons:

(i) They are the counterpart of encapsulation, i.e., they represent what is made public, while encapsulation represents what is hidden.
(ii) Compositionality requires that the services of a component are defined in terms of those of its sub-components.

To represent components and their interfaces in our general meta-model, we assume that DS has a signature including the following sorts and declarations (Decls):

$$\begin{array}{ll} \text{Sorts:} & \mathit{Request}, \mathit{Result}, \mathit{ReqType}, \mathit{ResType}, \mathit{OpType}, \mathit{Comp} \\ \text{Decls:} & >> \,:\, \mathit{ReqType} \times \mathit{ResType} \rightarrow \mathit{OpType} \\ & ::\,:\, \mathit{Comp} \times \mathit{OpType} \\ & \in\,:\, \mathit{Request} \times \mathit{ReqType} \;\mid\; \mathit{Result} \times \mathit{ResType} \end{array}$$

$\mathit{Request}$ is the sort of possible requests, and Result is that of possible results. $\mathit{ReqType}$ is the sort of request types, and $\mathit{ResType}$ is that of the result types. By the overloaded membership relation $\in$, each request type $Q : \mathit{ReqType}$ is interpreted as a set of requests, and each result type $R : \mathit{ReqType}$ as a set of results. Comp is the sort of possible components. OpType is introduced to represent component interfaces. It is freely generated by the constructor $>>$, i.e., its elements are of the form $Q >> R$, with $Q : \mathit{ReqType}$ and $R : \mathit{ResType}$. Interfaces are represented by the interface relation:

$$C :: Q >> R$$

It means that the component C accepts requests $q \in Q$ and yields results $r \in R$. We say that C supports the *operation type* $Q >> R$. The *interface* of C is the set of operation types it supports.

Requests and result types are disjoint. Thus a component cannot answer a request by another request to another component, that is, computation encapsulation is enforced. This will be further discussed in Section 4.3. $Request$, $Result$, $ReqType$, and $ResType$ are *open*, i.e., they depend on the DS at hand.

Example 1. In a programming language L, an operation template such as, e.g., $sum(x : int, y : int) : int$ can be represented in our formalism as the operation type

$$sum(x : int, y : int) >> int$$

where the request type $sum(x : int, y : int)$ represents all the call-instances $sum(m, n)$ and the result type is the data type int. That is, $Request$ is the set of all possible call-instances of L, $Result$ is the set of all the elements of the data types of L, $ReqType$ is the set of call-templates, $ResType$ is the set of data types of the language, and an operation type $m(x_1 : T_1, \ldots, x_n : T_n) >> T$ corresponds to an operation template $m(x_1 : T_1, \ldots, x_n : T_n) : T$.

Request and result types could also include semantic information, allowing us to use them as specifications and to deal with the correctness issue. Moreover, it may be useful to allow structured information, as illustrated by the following example.

Example 2. We introduce atomic request and result types by templates. A template is of the form $s(x_1 : T_1, \ldots, x_n : T_n)$, where s is the template symbol, $n \geq 0$, and $x_1 : T_1, \ldots, x_n : T_n$ are its parameters. We distinguish between request and result templates. In general, request templates correspond to procedure calls, such as $read()$, $write(x : string)$, etc. Result templates are semantic properties, such as $odd(x : int)$, $x = a + b$, etc. Semantic properties can also be used in requests, as pre-conditions. A request is an instance of a request template, e.g. $read$, $write(4)$, $odd(3)$, and so on. A result is an instance of a result template, e.g. $odd(5)$, $done$, and so on.

An example of an operation type using templates is:

$$sum(x : int, y : int) >> z = x + y$$

The meaning is that for every request $sum(m, n)$ we get a result $z = m + n$.

It may be useful to introduce structured templates, as shown by the following example.

$$read(F : text) >> x : int | notAnInteger$$

expresses the fact that reading from F yields an integer x, unless the characters read do not represent an integer. The use of structured templates is also useful in correspondence with connectors, as will be explained in Section 4.4.

4.2 Composition Operators

Composite components are built by means of composition connectors, starting from atomic components. The latter are built from computation units, by means of invocation connectors. To model this situation, we enrich the signature introduced in the previous section by adding:

Sorts: $\mathit{Unit}, \mathit{InvConn}, \mathit{CompConn}, \mathit{List}(X)$
Decls: The usual operations and relations on lists
$\bullet : \mathit{InvConn} \times \mathit{Unit} \to \mathit{Comp}$
$\bullet : \mathit{CompConn} \times \mathit{List}(\mathit{Comp}) \to \mathit{Comp}$
$\mathit{ctype} : \mathit{CompConn} \times \mathit{List}(\mathit{OpType}) \times \mathit{OpType}$

Unit is the sort of *units*, *InvConn* that of *invocation connectors*, and *CompConn* that of *composition connectors*. Parametric lists $\mathit{List}(X)$ are defined as usual, and a list will be represented by the notation $[x_1, \ldots, x_n]$. The overloaded operator $\bullet$ is the composition operator.

Components are defined by *composition terms*. A composition term T indicates how a component is built by connectors starting from units or components already defined and stored in the repository of the DS. Composition terms are generated by the composition $\bullet$: $i \bullet u$ denotes the application of an invocation connector i to a unit u, and $k \bullet [T_1, \ldots, T_n]$ denotes the application of a composition operator k to the sub-terms (denoting sub-components) $T_1, \ldots, T_n$.

Composition connectors are typed by *ctype*. We will write $k : [Op_1, \ldots, Op_n] \to Op$ as a different notation for $\mathit{ctype}(k, [Op_1, \ldots, Op_n], Op)$. If k has a composition type $Op_1, \ldots, Op_n \to Op$, then $k \bullet [T_1, \ldots, T_n]$ has operation type Op whenever it is applied to $T_1 :: Op_1$, …, $T_n :: Op_n$.

Not all the composition terms represent *components*. Components, together with their interface relation ::, are defined by inductive *composition rules* of the following form:

$$\frac{}{i \bullet u :: Op}\ r(i,u)$$

$$\frac{T_1 :: Op_1 \quad \ldots \quad T_n :: Op_n}{k \bullet [T_1, \ldots, T_n] :: Op}\ r(k)$$

For the invocation connector rule $r(i, u)$, the operation type Op is determined by both i and u. For the composition connector rule $r(k)$, Op is determined by the composition type $Op_1, \ldots, Op_n \to Op$ of k. Connectors and the related composition rules are, in general, domain specific and depend on the DS. The components of a DS are defined as follows:

Definition 3. A *composition term* T is a component of a DS with operation type Op iff $T :: Op$ can be derived by the composition rules of the DS.

New composite components can be introduced in the repository of the DS by definitions of the form $C := T$, where C is a new constant symbol and T is a composition term. As usual, the definition $C := T$ expands the current signature by the new constant $C : \mathit{Conn}$ and introduces the definition axiom $C = T$. The interface relation of a component C introduced by a definition $C := T$ is that of T:

$$\text{if } C := T, \text{ then } C :: Q >> R \text{ iff } T :: Q >> R \tag{1}$$

Typed composition connectors have a compositional semantics given by:

– the composition rules of the DS;
– the execution rules explained in Section 4.3, which give the run-time semantics of a component in terms of that of its sub-components.

We distinguish between designing a DS and using it. *Designing* a DS means designing its units and, more importantly, the rules of its connectors, according to the general assumptions of the meta-model. *Using* a DS to design systems and components means using its units and its composition rules. Although, in general, connectors and their composition rules are domain specific, there are general-purpose connectors. Some of them will be shown in Section 4.4. In the following example we show the general-purpose connector $pipe$.

Example 3. A pipe connector $pipe : [Q_1 >> R_1, Q_2 >> R_2] \to Q_1 >> R_2$ assumes a map $p : Result \to Request$, to pipe the results $r_1 \in R_1$ of component $C_1 :: Q_1 >> R_1$ into requests $q_2 \in Q_2$ for $C_2 :: Q_2 >> R_2$ (details in Example 4). The composition rule is:

$$\frac{C_1 :: Q_1 >> R_1 \quad C_2 :: Q_2 >> R_2}{pipe \bullet [C_1, C_2] :: Q_1 >> R_2}\, r(pipe).$$

4.3 A Run-Time Semantics

As mentioned before, to run a component represented by a composition term T, we need to compile the units and connectors of T into binaries, to deploy binaries according to T, to load them into the memory and to launch them. This process requires an appropriate infrastructure, that guarantees that the implementation agrees with the *intended run-time semantics* of T. In this section we define the intended run-time semantics in an abstract, i.e. implementation independent, way. To this end, we enrich our signature as follows:

Sorts: $D, Instance, Step$
Decl: $halt, error : D$
$\mapsto :\ D \times Request \times D \times Result \to Step;$
$\mathrm{i}\ :\ Comp \times D \to Instance;$
$exec\ :\ Comp \times Step$

Instance is the sort of *run-time instances*, and D is the sort of *data* that can be contained in the memory of instances. We do not model data and their structure in this paper. In the examples we will assume that D is closed with respect to the pairing operation (i.e., if $d_1, d_2 \in D$, then $\langle d_1, d_2 \rangle \in D$). By $\mathrm{i}(C, d)$ we represent a running instance of a component C with current memory content $d \in D$. The relation $data(C, d)$ indicates which data are admitted for a component C.

Step is the sort of *execution steps*. It is freely generated by $\mapsto$, i.e., its elements are uniquely represented by terms of the form $\mapsto (d, q, d', r)$. A term $\mapsto (d, q, d', r)$ is also written $[d, q] \mapsto [d', r]$. It indicates an execution step from the current memory content d and request q, into the new memory content d' and result r.

Instances can execute requests. Let $\mathrm{i}(C, d)$ be a run-time instance with operation type $C :: QT >> RT$, and let $q \in Q$ be a request. The *execution relation* of a component C

$$exec(C, [d, q] \mapsto [d', r])$$

indicates that when the instance $\mathrm{i}(C, d)$ executes a request q, it performs the execution step $[d, q] \mapsto [d', r]$. To treat regular halting and run-time errors, we consider $halt$ and $error$ as particular memory contents:

$$exec(C, [d, q] \mapsto [halt, r])$$
$$exec(C, [d, q] \mapsto [error, r])$$

We define the run-time semantics of an atomic component $i \bullet u$ by a map $\mathcal{M}(i, u) : D \times Request \to D \times Result$ as follows:

$$exec(i \bullet u, [d, q] \mapsto [d', r]) \ \leftrightarrow \ \mathcal{M}(i, u)(d, q) = \langle d', r\rangle$$

We define the run-time semantics of a non-atomic component $k \bullet [T_1, \dots, T_n]$ by a map $\mathcal{M}(k) : Step^n \to Step$ as follows:

$$\begin{aligned} exec(k \bullet [T_1, \dots, T_n], S) \leftrightarrow & \wedge_{j=1}^{n} exec(T_j, S_j) \wedge \\ & S = \mathcal{M}(k)(S_1, \dots, S_n) \end{aligned}$$

Invocation connectors provide *encapsulation* for atomic components through interfaces. The unit u in an atomic component $i \bullet u$ cannot directly call any other unit or component. It can only provide results, i.e., the only way of requiring a service from outside (if needed) is to pass the request as a result through the invocation connector. This "request-result" is then managed by the other connectors, that is, control is performed by connectors. The semantics of composition connectors is compositional. Indeed: (a) it preserves encapsulation through interfaces and (b) $\mathcal{M}(k)$ indicates how the resulting step S is obtained from the computation steps S_j of the sub-components, in a way that does not depend on the specific features of the sub-components, but only on their operation types $Op_1, \dots, Op_n$ and on the connector $k : [Op_1, \dots, Op_n] \to Op$.

An abstract compositional run-time semantics is useful for two main reasons. The first one is that a compositional semantics supports "predictability", since the result of a composition is also a component and its services are defined in terms of those of the sub-components. The second reason is that it abstracts from the implementation details, related to the compilation of composition terms into runnable binaries. The correctness of different implementations with respect to the abstract run-time semantics fixed for composition terms supports interoperability. Thus, *designing* the abstract run-time semantics of connectors and units means defining the maps $\mathcal{M}(i, u)$ and $\mathcal{M}(k)$ according to the general requirements explained above. *Implementing* it means implementing a run-time infrastructure that is correct with respect to the abstract semantics.

The correctness of an implementation is with respect to the abstract execution semantics. With the step $[d, q] \mapsto [d', r]$ we associate the *observable step* $q \mapsto r$. An implementation is correct if the observable steps obtained by running it coincide with those defined by the abstract run-time semantics. That is, we abstract from the internal representation of data, and we are only interested in observable requests and results.

Example 4. Here we show the run-time semantics of the $pipe$ rule of Example 3.

$$\begin{aligned} exec(pipe \bullet [C_1, C_2], [\langle d_1, d_2\rangle, q_1], [\langle d'_1, d'_2\rangle, r_2]) \leftrightarrow\ & exec(C_1, [d_1, q_1] \mapsto [d'_1, r_1])\wedge \\ & exec(C_2, [d_2, q_2] \mapsto [d'_2, r_2])\wedge \\ & q_2 = p(r_1) \end{aligned}$$

In this rule, results $r_1 \in R_1$ are piped into requests $p(r_1) \in Q_2$, the sub-component C_1 has data d_1 and the sub-component C_2 has separate data d_2, and the whole component has data $\langle d_1, d_2\rangle$ (i.e., $data(pipe \bullet [C_1, C_2], d)$ holds iff $d = \langle d_1, d_2\rangle$, with $data(C_1, d_1)$ and $data(C_2, d_2)$). We may have different kinds of pipe, e.g., the piping mechanism could also depend on the request q_1.

4.4 The Bank Example

In this section, we illustrate our general model. Firstly we outline part of a possible DS, and then we apply it to the bank example (Section 3.4).

The DS defines interfaces through structured templates, as in Example 2. Here we consider the structuring operators $|$, sel and *, defined as follows.

- A request/result of a type $A_1|\cdots|A_n$ is a pair $\langle k, a\rangle$, with $1 \leq k \leq n$ and $a \in A_k$.
- A request/result of type $sel(p \in S : A(p))$ is a pair $\langle v, a\rangle$, where S is a finite set of values, $v \in S$ and $a \in A(v)$.
- A request/result of type A^* is a sequence $[a_1, \ldots, a_n]$ such that $a_i \in A$.

The composition rules for the connectors related to the above structures are:

$$\frac{C :: Q >> R}{loop \bullet [C] :: Q^* >> R^*}\ r(loop) \qquad \frac{C_1 :: Q_1 >> R_1 \quad \ldots \quad C_n :: Q_n >> R_n}{case \bullet [C_1, \ldots, C_n] :: Q_1|\cdots|Q_n >> R}\ r(case)$$

$$\frac{C(v_1) :: Q(v_1) >> R \quad \ldots \quad C(v_n) :: Q(v_n) >> R}{sel \bullet [C(v_1), \ldots, C(v_n)] :: sel(p \in \{v_1, \ldots, v_n\} : Q(p) >> R)}\ r(sel)$$

The execution semantics is:

$$\begin{aligned} exec(case \bullet [C_1, \ldots, C_n], [d, \langle j, q\rangle] \mapsto [d', r]) &\leftrightarrow exec(C_j, [d_j, q] \mapsto [d'_j, r'])\wedge \\ & \quad (\wedge_{k\neq j}\ d'_k = d_k) \wedge r' \overset{R_j,R}{\mapsto} r \\ exec(sel \bullet [C(v_1), \ldots, C(v_m)], [d, \langle v_j, q\rangle] \mapsto [d', r]) &\leftrightarrow exec(C(v_j), [d_j, q] \mapsto [d'_j, r]) \\ & \quad \wedge(\wedge_{k\neq j}\ d'_k = d_k) \\ exec(loop \bullet C, [d, [q|\underline{q}]] \mapsto [d', [r|\underline{r}]]) &\leftrightarrow exec(C, [d, q] \mapsto [d_1, q_1])\wedge \\ & \quad exec(loop \bullet C, [d_1, \underline{q}], [d', \underline{r}]) \end{aligned}$$

The $case$ component has $data(case \bullet [C_1, \ldots, C_n], \langle d_1, \ldots, d_n\rangle)$, with $data(C_j, d_j)$ (similarly for the sel component). The connector $case$ requires that there is a map $r' \overset{R_j,R}{\mapsto} r$ from $r' \in R_j$ into $r \in R$, depending on the result types R_j and R. In particular:

$$a \overset{A,A|B}{\mapsto} \langle 1, a\rangle$$
$$b \overset{B,A|B}{\mapsto} \langle 2, b\rangle$$

The connector sel applies to n instances of a parametric component $C(p) :: Q(p) >> R$ and executes the one indicated by v_j. The $loop$ connector iterates C over a sequence of requests. Besides these connectors, we also have the pipe connector explained above.

Now we sketch a possible construction of the bank system (Section 3.4) using the DS partially outlined above.

In Fig. 9, the invocation connectors for the ATM and bank computation units encapsulate them into atomic components with the following operation types:

$$\begin{array}{l} atmC := IA \bullet ATM \;\; :: \mathit{choose}() >> (\mathit{chosen}(n, \mathit{Acc}, \mathit{Op}) | \mathit{notOkPin}) \\ b(n) := IB_n \bullet \mathit{Bank}_n :: \mathit{do}(n, \mathit{Acc}, \mathit{Op}) >> \mathit{amount}(\mathit{Acc}, A) | \mathit{refusedOp} \end{array}$$

Firstly, we informally explain the semantics of the atomic components.

In the operation type of $atmC$, $choose()$ indicates that the user inputs a PIN and an operation choice. If the PIN is not recognised, the result is $notOkPin$, otherwise it is $chosen(n, Acc, Op)$, indicating that the PIN corresponds to the account Acc of the bank number n, and Op is the operation chosen.

In the operation type of $b(n)$, $do(n, Acc, Op)$ indicates that an operation Op has been requested on the account Acc of bank n. The operation Op may be accepted or refused, as indicated by the result type of $amount(Acc, A)|refusedOp$. If accepted, the result $amount(Acc, A)$ indicates that A is the amount of Acc after the operation. The operation (when not refused) may update the current amount.

Now we compose the atomic components by connectors, to obtain our system. We firstly build the *banks* composite of the two banks and sending a requested operation to the target bank n, by means of a selector connector:

$$\begin{array}{l} \mathit{banks} := \mathit{sel} \bullet [b(1), b(2)] \;\; :: \\ \mathit{sel}(n \in \{1, 2\} : \mathit{do}(n, \mathit{Acc}, \mathit{Op})) >> (\mathit{amount}(\mathit{Acc}, A) | \mathit{refusedOp}) \end{array}$$

By a $pipe$, we build the component $atmOp$, performing a single ATM request and, by a $loop$, the component $system$, looping on ATM requests, as follows.

$$\begin{array}{rl} \mathit{atmOp} & := \mathit{pipe} \bullet [\mathit{atmC}, \mathit{case} \bullet [\mathit{banks}, \mathit{noOperation}]] :: \\ & \quad \mathit{choice}() >> (\mathit{amount}(\mathit{Acc}, A) | \mathit{refusedOp}) | \mathit{notOkPin} \\ \mathit{system} & := \mathit{loop} \bullet \mathit{atmOp} :: \mathit{choice}()^* >> ((\mathit{amount}(\mathit{Acc}, A) | \mathit{refusedOp}) | \mathit{notOkPin})^* \end{array}$$

The internal connector $case \bullet [C,\ noOperation]$ is to be considered as a part of the $pipe$ connector, and the $noOperation$ branch is not a sub-component. It maps results into results and is used to bypass C. In our example, the request type of $case \bullet [banks, noOperation]$ is $sel(n \in \{1, 2\} : do(n, Acc, Op))|notOkPin$. If the result of $atmC$ is $\langle 2, notOkPin \rangle$, we pipe it into $\langle 2, notOkPin \rangle$ itself, so that $case$ passes the result $notOkPin$ to the $noOperation$ branch. If the result of $atmC$ is $\langle 1, chosen(n, Acc, Op\ ) \rangle$, we pipe it into $\langle 1, \langle n, do(n, Acc, Op) \rangle \rangle$, so that the request $\langle n, do(n, Acc, Op) \rangle$ is passed to *banks*.

To illustrate the run-time semantics of connectors, we show the execution of a request. The whole system has the following data:

– a database $atmdb$, associating each valid PIN to a bank and an account number;
– databases db_i (with $i = 1$ or $i = 2$), containing the accounts of bank i.

The data-components association should be decided in the deployment phase. Since here we abstract from it, data are triples $d = \langle atmdb, db_1, db_2\rangle$, where $atmdb$ is used by $atmC$, db_1 by $b(1)$, and db_2 by $b(2)$.

By the semantics of $loop$, the computation step corresponding to a sequence of requests of length n has the form

$$[[d_0, [choice(), choice(), \ldots, choice()]] \mapsto [d_n, [Res_1, Res_2, \ldots, Res_n]]]$$

where each $[d_{n-1}, choice()] \mapsto [d_n, Res_n]$ is performed by $atmOp$. We consider the first step $[d_0, choice()] \mapsto [d_1, Res_1]$, and we assume that the the user inputs a correct PIN and requires a withdrawal of £50, and that the (correct) PIN input is related to the bank $b(2)$ and the account number $Acc = 2341$. We assume that the total amount of the account (stored in db_2) is £5170.

By the semantics of the pipe connector, we have two sub-steps:

$$[d_0, choice()] \overset{atmC}{\mapsto} [d_0, \langle 1, chosen(2, 2341, withdraw(50))\rangle]$$
$$[d_0, \langle 1, \langle 2, op(2, 2341, withdraw(50))\rangle\rangle] \overset{case\bullet[banks,noOperation]}{\mapsto} [d_1, Res_1]$$

where we indicate on the top of $\mapsto$ the sub-component performing the step. The first step corresponds to the semantics of the atomic component $atmC$ informally explained above, and its result $\langle 1, chosen(2, 2341, withdraw(50))\rangle$ is piped into the request for the second step. By the semantics of $case$, the second step is obtained from the sub-step:

$$[d_0, \langle 2, op(2, 2341, withdraw(50))\rangle] \overset{sel\bullet[b(1),b(2)]}{\mapsto} [d_1, Res_1]$$

By the semantics of sel, the latter is obtained from the step

$$[d_0, op(2, 2341, withdraw(50))] \overset{b(2)}{\mapsto} [d_1, \langle 1, amount(2341, 5120)\rangle]$$

performed by the atomic component $b(2)$, which updates the current amount of the account 2341 stored in the database db_2. By the semantics of $case$, the result Res_1 is obtained by the mapping $\overset{R_2,R}{\mapsto}$. Here R_2 is $amount(Acc, A)|refusedOp$, and R is $(amount\ (Acc, A)|refusedOp)|notOkPin$. Thus, the mapping is:

$$\langle 1, amount(2341, 5120)\rangle\rangle \overset{R_2,R}{\mapsto} Res_1 = \langle 1, \langle 1, amount(2341, 5120)\rangle\rangle$$

The result $Res_1 : (amount(Acc, A)|refusedOp)|notOkPin$ indicates that the pin is okay, the operation has been performed successfully and the new amount is £5120.

5 Discussion

In our component model, exogenous connectors play a fundamental role, not only for constructing atomic components but also for composing components into composites, whilst providing interfaces to all these (atomic and composite) components. Independently, exogenous connection has been defined as exogenous coordination in coordination languages for concurrent computation [2]. Also independently, in object-oriented programming, the courier patter [10] uses the idea of exogenous connection. There are also similarities with Service Oriented Architectures [29], where business processes accessing (independent) services can be specified by means of an orchestration language. However, no current model relies on encapsulation requirements as strong as ours. We believe that strong encapsulation is a key feature to obtain truly independent and reusable components.

The preliminary formalisation of our component model provides a semantic framework for our approach to component-based software development. Our model and formalisation highlight the basic ideas and fix the minimal requirements for a component system based on exogenous connectors, whilst leaving completely open the choice of the specific connectors and of the interface language. The possibility of designing connectors and interfaces in our model is illustrated by the example of Section 4.4, which outlines part of a possible DS. The example shows that an interface language tailored to the structural properties of connectors allows us to link the meaning of data involved in computation to the structure of components. In this way, the semantic framework of our model should enable us to reason formally not only about the correctness of individual atomic components, but also about the correctness of any composite component, and therefore the correctness of any system built from components.

Consequently, two benefits should accrue, viz. *predictable assembly* of component-based systems, and *verified software* built from pre-verified components. Predictable assembly is the ultimate goal of Component-based Software Engineering, whilst verified software has remained a grand challenge for a long time [13]. We believe that our component model can contribute to predictable assembly because it allows us to generate interfaces to any composite component (or system) we build, directly from the interfaces of its constituent (sub)components.

By the same token, our model can contribute to the verified software challenge by breaking the problem down into smaller sub-problems, and in particular by enabling proof reuse, i.e. using proofs of sub-components directly in the proof of a composite or system. To realise these benefits, we are implementing our component model in the Spark [4] language, which has proof tools which can support verification of components.

Our formalisation (and model) is only preliminary at present however. Many issues still need to be investigated, e.g. what kinds of connectors are useful in practice, considering the constructs introduced in other approaches, e.g. in web service orchestration languages such as BPEL [1]. The problem is to establish whether particular connectors are compositional and preserve strong encapsulation.

For instance, in the bank example, we have used a loop connector at the outer-most level, simply because it is natural to use such a connector to handle continuous inputs from clients. This connector, as defined here, is compositional because it iterates on a finite sequence of requests. Ideally, however, it should allow an infinite stream of inputs, but unfortunately such a loop connector is not compositional. Clearly whether a loop terminates is usually only known at run-time. So whether it can ever be used as a composition connector at design time remains a moot point. Equally, a non-terminating loop connector may be acceptable, even desirable, at the outer-most level. It would be interesting to study the possibility of introducing infinite streams into our approach while maintaining a notion of control encapsulation, by using general formal contexts, such as FOCUS [7].

6 Conclusion

In this paper, we have presented a software component model and its preliminary formalisation. Encapsulation and compositionality are the key concepts that underlie our model. In contrast, existing component models tend to use either objects or architectural units as components, which are neither well encapsulated nor compositional.

Our component model is based on exogenous connectors. Using these connectors to construct and compose components is the key to achieving encapsulation and compositionality. Composite components constructed by exogenous connectors are self-similar, which makes a compositional approach to system construction possible. In contrast to existing software component models, our self-similar components are also encapsulated and compositional.

Another benefit of exogenous connection is that components are loosely coupled, since control is originated and encapsulated by connectors, unlike ADL connectors that do not originate or encapsulate control. As a result, systems are modular and therefore easier to maintain and re-configure.

References

1. T. Andrews, F. Curbera, H. Dholakia, Y. Goland, J. Klein, F. Leymann, K. Liu, D. Roller, D. Smith, S. Thatte, I. Trickovic, and S. Weerawarana. *Business Process Execution Language for Web Services(BPEL4S) - Version 1.1*. IBM, http://www-106.ibm.com/developerworks/library/ws-bpel/, 2004.
2. F. Arbab. The IWIM model for coordination of concurrent activities. In P. Ciancarini and C. Hankin, editors, *LNCS 1061*, pages 34–56. Springer-Verlag, 1996.
3. C. Atkinson, J. Bayer, C. Bunse, E. Kamsties, O. Laitenberger, R. Laqua, D. Muthig, B. Paech, J. Wüst, and J. Zettel. *Component-based Product Line Engineering with UML*. Addison-Wesley, 2001.
4. J. Barnes. *High Integrity Software: The SPARK Approach to Safety and Security*. Addison-Wesley, 2003.
5. D. Box. *Essential COM*. Addison-Wesley, 1998.
6. M. Broy, A. Deimel, J. Henn, K. Koskimies, F. Plasil, G. Pomberger, W. Pree, M. Stal, and C. Szyperski. What characterizes a software component? *Software – Concepts and Tools*, 19(1):49–56, 1998.
7. M. Broy and K. Stølen. *Specification and Development of Interactive Systems: Focus on Streams, Interfaces, and Refinement*. Springer, 2001.
8. E. Bruneton, T. Coupaye, and M. Leclercq. An open component model and its support in Java. In *Proc. 7th Int. Symp. on Component-based Software Engineering*, pages 7–22. Springer -Verlag, 2004.
9. J.L. Fiadeiro, A.Lopes, and M.Wermelinger. A mathematical semantics for architectural connectors. LNCS 2793, pages 178-221, 2003.
10. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. The Courier pattern. *Dr. Dobb's Journal*, Feburary 1996.
11. D. Garlan, R.T. Monroe, and D. Wile. Acme: Architectural description of component-based systems. In G.T. Leavens and M. Sitaraman, editors, *Foundations of Component-Based Systems*, pages 47–68. Cambridge University Press, 2000.
12. G.T. Heineman and W.T. Councill, editors. *Component-Based Software Engineering: Putting the Pieces Together*. Addison-Wesley, 2001.

13. IFIP TC2 working conference on Verified Software: Theories, Tools, Experiments, 10-13 October 2005, ETH Zürich, Switzerland. `http://vstte.ethz.ch/`.
14. J. Ivers, N. Sinha, and K.C Wallnau. A Basis for Composition Language CL. Technical Report CMU/SEI-2002-TN-026, CMU SEI, 2002.
15. K.-K. Lau and M. Ornaghi. Specifying compositional units for correct program development in computational logic. In M. Bruynooghe and K.-K. Lau, editor, *Program Development in Computational Logic, Lecture Notes in Computer Science 3049*, pages 1–29. Springer-Verlag, 2004.
16. K.-K. Lau, P. Velasco Elizondo, and Z. Wang. Exogenous connectors for software components. In *Proc. 8th Int. SIGSOFT Symp. on Component-based Software Engineering, LNCS 3489*, pages 90–106, 2005.
17. K.-K. Lau and Z. Wang. A survey of software component models. Pre-print CSPP-30, School of Computer Science, The University of Manchester, April 2005. `http://www.cs.man.ac.uk/cspreprints/PrePrints/cspp30.pdf`.
18. K.-K. Lau and Z. Wang. A taxonomy of software component models. In *Proc. 31st Euromicro Conference*, pages 88–95. IEEE Computer Society Press, 2005.
19. N.R. Mehta, N. Medvidovic, and S. Phadke. Towards a taxonomy of software connectors. In *Proc. 22nd Int. Conf. on Software Engineering*, pages 178–187. ACM Press, 2000.
20. B. Meyer. The grand challenge of trusted components. In *Proc. ICSE 2003*, pages 660–667. IEEE, 2003.
21. Sun Microsystems. Enterprise Java Beans Specification, Version 3.0, 2005.
22. O. Nierstrasz, G. Arévalo, S. Ducasse, R. Wuyts, A. Black, P. Müller, C. Zeidler, T. Genssler, and R. van den Born. A component model for field devices. In *Proc. 1st Int. IFIP/ACM Working Conference on Component Deployment*, pages 200–209. ACM Press, 2002.
23. OMG. *UML 2.0 Superstructure Specification*. `http://www.omg.org/cgi-bin/doc?ptc/2003-08-02`.
24. OMG. *CORBA Component Model, V3.0*, 2002. `http://www.omg.org/technology/documents/formal/components.htm`.
25. F. Plasil, D. Balek, and R. Janecek. SOFA/DCUP: Architecture for component trading and dynamic updating. In *Proc. ICCDS98*, pages 43–52. IEEE Press, 1998.
26. M. Shaw and D. Garlan. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice Hall, 1996.
27. Sun Microsystems. *JavaBeans Specification*, 1997. `http://java.sun.com/products/javabeans/docs/spec.html`.
28. C. Szyperski, D. Gruntz, and S. Murer. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley, second edition, 2002.
29. E. Thomas. *Service-Oriented Architecture: Concepts, Technology, and Design*. Prentice Hall, 2005.
30. R. van Ommering, F. van der Linden, J. Kramer, and J. Magee. The Koala component model for consumer electronics software. *IEEE Computer*, pages 78–85, March 2000.

Appendix

We formalise our model as an open specification framework [15]. We distinguish between *open* and *defined* symbols. The meaning of the defined symbols is established by the definition axioms, in terms of the open ones. The open symbols are to be axiomatised when designing a specific DS based on exogenous connectors. The constraint axioms represent proof obligations to be satisfied when axiomatising a DS. The axiomatisation presented here contains the minimal requirements and has a loose semantics.

The signature is the one explained in Section 4. We import the parametric abstract data type $List(X)$. The defined sorts are $OpType$, $List(X)$, $Comp$, $Instance$, and $Step$. They are freely generated from the open sorts, according to the following constructor axioms (see [15]):

$$\begin{array}{lll} OpType & \text{constructed by} & >>: ReqType \times ResType \to OpType; \\ Comp & \text{constructed by} & \bullet : InvConn \times Unit \to Comp, \\ & & \bullet : CompConn \times List(Comp) \to Comp; \\ Instance & \text{constructed by} & \mathrm{i} : Comp \times D \to Instance; \\ Step & \text{constructed by} & \mapsto: D \times Request \times D \times Result \to Step; \end{array}$$

In Section 4.3 we have informally introduced the semantic function $\mathcal{M}$. Here we introduce it in the signature by the declaration

$$\mathcal{M} : InvConn \times Unit \times Step \mid CompConn \times List(Step) \times Step$$

and we axiomatise $exec$ by mutual recursion (axioms $dax_{1,j}$), using the auxiliary (overloaded) predicate $exec : List(Comp) \times List(Step)$. The other axioms introduce auxiliary predicates used later, in the constraint axioms: dax_2 extend the overloaded interface relation to sequences of components ($[C_1, \ldots, C_n] :: [Op_1, \ldots, Op_n]$ indicates that $C_i :: Op_i$); $dax_{3,k}$ introduce the relations $domain$, $range$ and $stype$, indicating the expected types of requests, data and results in computation steps; $dax_{4,1}$ introduce $total(C, Op)$, indicating that $C :: Op$ computes a total input-output relation, and the axiom $dax_{4,2}$ extends $total$ to lists of components. In the axioms we will use the following typed variables: $i : InvConn$, $k : CompConn$, $u : Unit$, $S : Step$, $LC : List(Comp)$, $LS : List(Step)$, $LO : List(Op)$, $Q : ReqType$, $R : ResType$, $q : Request$, $r : Result$, $d : D$, $Op : OpType$, and we will leave the most external quantification understood.
Definition Axioms:

$$\begin{array}{ll} dax_{1,1} & exec(i \bullet u, S) \;\leftrightarrow\; \mathcal{M}(i, u, S) \\ dax_{1,2} & exec(k \bullet LC, S) \;\leftrightarrow\; \exists LS(exec(LC, LS) \land \mathcal{M}(k, LS, S)) \\ dax_{1,3} & exec([], []) \;\land\; (exec([C|LC], [S|LS]) \;\leftrightarrow\; exec(C, S) \land exec(LC, LS)) \\ dax_2 & [] :: [] \;\land\; ([C|LC] :: [Op|LO] \;\leftrightarrow\; C :: Op \land LC :: LO) \\ dax_{3,1} & domain(d, q, C, Q >> R) \;\leftrightarrow\; C :: Q >> R \land q \in Q \land data(C, d) \\ dax_{3,2} & range(d, r, C, Q >> R) \;\leftrightarrow\; C :: Q >> R \land r \in R \land (data(C, d) \lor d = halt) \\ dax_{3,3} & stype([d, q] \mapsto [d', r], C, Op) \;\leftrightarrow\; domain(d, q, C, Op) \land range(d', r, C, Op) \\ dax_{3,4} & stype([], [], []) \land \\ & (stype([S|LS], [C|LC], [Op|LO]) \leftrightarrow stype(S, C, Op) \land stype(LS, LC, LO)) \\ dax_{4,1} & total(C, Op) \;\leftrightarrow\; C :: Op \land \\ & \forall d, q(domain(d, q, C, Op) \to \exists d', r\; exec(C, [d, q] \mapsto [d', r])) \\ dax_{4,2} & total([], []) \;\land\; (total([C|LC], [Op|LO]) \;\leftrightarrow\; total(C, Op) \land total(LC, LO)) \end{array}$$

In the following, instead of $ctype(k, LO, O)$ and $stype(S, C, Op)$ we will use the more intuitive notation $k : LO \to Op$ and $S : (C :: OP)$. Now we give the constraint axioms. By c_1 we require that a composition term $k \bullet [T_1, \ldots, T_n]$ is a component with operation type Op *only if* the subcomponents $T_1, \ldots, T_n$ agree with the type of k. The if part is left open and has to be fixed by the composition rules of the specific DS

(see Definition 3). By $c_{2,i}$ we require that the semantic relation $\mathcal{M}$ conforms to the domain and range types of components. The other constraints allow us to prove Theorem 1, which states that each component terminates.

Constraints:

c_1 $(k \bullet L) :: Op \rightarrow \exists\, LO(L :: LO \wedge k : LO \rightarrow Op)$

$c_{2,1}$ $(i \bullet u) :: Op \ \rightarrow\ \forall\, S(\mathcal{M}(i,u,S) \rightarrow S : (i \bullet u :: Op))$

$c_{2,2}$ $(k \bullet LC :: Op) \wedge LS : (LC :: LO) \rightarrow \forall S(\mathcal{M}(k, LS, S) \rightarrow S : (k \bullet LS :: Op))$

$c_{3,1}$ $(i \bullet u) :: Op \rightarrow \forall\, d, q(domain(d,q,i \bullet u, Op) \rightarrow \exists\, d', r\ \mathcal{M}(i,u,[d,q] \mapsto [d',r]))$

$c_{3,2}$ $(k \bullet LC :: Op) \wedge total(LC, LO)\ \rightarrow\ \forall\, d, q\,(domain(d,q,k \bullet LC, Op)$
$\rightarrow\ \exists\, d', r(exec(k \bullet LC, [d,q] \mapsto [d',r])))$

Theorem 1. *The following sentences can be proved from the above axioms:*
$\forall C, Op(C :: Op \rightarrow total(C, Op))$

It is worthwhile to remark that constraints are proof obligations when a specific DS is axiomatised. In particular, c_1 is a proof obligation for the composition rules, and the other constraints are proof obligations for the relation $\mathcal{M}$ defining the run-time semantics. We show how such proof obligations work by proving, as an example, that the semantics for $pipe(Q_1 >> R_1, Q_2 >> R_2)$ satisfies $c_{3,2}$.

Let $pipe \bullet [C_1, C_2]$ be a generic pipe component, and $[\langle d_1, d_2\rangle, q_1]$ be a generic element of its domain. We have to prove that there is a step

$$exec(pipe \bullet [C_1, C_2], [\langle d_1, d_2\rangle, q_1] \mapsto [\langle d_1', d_2'\rangle, r_2])$$

By the assumption $total([C_1, C_2] :: [Q_1 >> R_1, Q_2 >> R_2])$ of $c_{3,2}$, we get $total(C_1 :: Q_1 >> R_1)$ and $total(C_2 :: Q_2 >> R_2)$. By $total(C_1 :: Q_1 >> R_1)$, there is $[d_1', r_1]$ such that $exec(C_1, [d_1, q_1] \mapsto [d_1', r_1])$. By the pipe operation we can build $q_2 = p(r_1)$, and we get $[d_2, q_2]$ in the domain of C_2. Finally, by $total(C_2 :: Q_2 >> R_2)$, we can conclude that the required result $[\langle d_1', d_2'\rangle, r_2]$ exists. Thus, the semantics of $pipe$ is well defined in our model.