A Language to Bridge the Gap between Component-based Design and Implementation

Luc Fabresse[1,*], Noury Bouraqadi[1], Christophe Dony[2], Marianne Huchard[2]

Abstract

Since years, multiple researches studied component-based software development. Nevertheless, most component-based software systems do not use components at implementation stage. We believe that one of the main causes is a lack of support for Component-Oriented Programming (COP). Indeed, most of proposed component models such as Unified Modeling Language (UML), Corba Component Model (CCM), Enterprise JavaBeans (EJB) are only available at design time. The fact that implementation relies on object-oriented languages which prevent developers from fully switching to COP. In this paper, we identify five important requirements (decoupling, adaptability, unplanned connections, encapsulation and uniformity) for COP based on an analysis of the state of the art and the limitations of existing work. We propose an extended version of the SCL component language that fulfills these requirements. A prototype of SCL and a concrete experiment validate this proposal.

Keywords: Components, Programming language, Unplanned connection, Encapsulation, Uniformity

1. Introduction

For decades, researches in component-based software development (CBSD) [McI68, Szy02] promote components off-the-shelf (COSTS) as a promising post-object paradigm. Inspired by the electronics industry, the goal is to assemble components – which are reusable software's pieces – to create applications. CBSD promotes standardisation for better reusability. It also rationalises software engineering by clearly stating roles and activities. Considering the implementation stage, component-oriented programming (COP) is at least twofold (cf. Figure 1): (i) programming reusable components (design *for* reuse achieved by a *component developer*) and (ii) programming an application by reusing, or connecting, or composing components (design *by* reuse achieved by an *application developer*) [Ous05].

Previous work [LW05, CSVC10] have introduced or adapted concepts such as *component*, *port*, *architecture*, *composite* and mechanisms such as *connection* or *composition*. Nevertheless, CBSD is still seldom used in practice to implement software systems. One of the reasons is that component-orientation has been more studied at design stage (e.g Architecture Description Languages (ADL) [SDK+95, MT00]) rather than implementation stage.

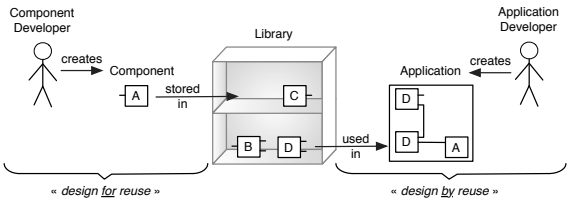

Figure 1: The Component-Oriented Programming Duality.

One of the reasons of the object-orientation success is surely because it brings a continuum between the way of thinking of designers at design stage and the way of programming of developers at implementation stage thanks to object-oriented languages. Analogously, we believe that *component-oriented languages* should support CSBD and provide the same continuum between design and implementation with components.

Nowadays, there are four approaches to use components at the implementation stage:

*Corresponding author
Email addresses: luc.fabresse@mines-douai.fr (Luc Fabresse), noury.bouraqadi@mines-douai.fr (Noury Bouraqadi), dony@lirmm.fr (Christophe Dony), huchar@lirmm.fr (Marianne Huchard)
[1]Université Lille Nord de France, Ecole des Mines de Douai, France
[2]Université Montpellier 2, LIRMM, France

1. Using a general purpose programming language (usually object-oriented) and relying on a set of conventions (such as Javabeans [Ham97]).

2. Extending an object-oriented framework (such as Julia/Fractal [BCL+06], EJB1.0 and EJB2.0 [MH99])

3. Using component specifications to partially generate OO code. These specifications can be written in multiple ways such as using ADLs or Java annotations as in EJB3.0 [BMH06]. This approach is related to model transformation because specifications can be refined iteratively to become more and more concrete.

4. Using a component-oriented language (COL) (such as ArchJava [ACN02], ComponentJ [SC00]).

Component related concepts vanish at the implementation stage when using a general purpose programming language. Framework-based approaches are better but also rely on the programmers discipline. Model transformation approaches bypass this limitation. Nevertheless, the component related concepts disappear in the generated code, making debugging problematic. The best approaches rely on a programming language with constructs that match concepts used when designing component-based software. But existing COLs still exhibit some open issues.

The aim of this paper is to propose a usable COL that provides a continuum between design and implementation with components.

The contributions of this article are multiple. First, we propose five requirements for COP based on the open issues detected in the state of the art. These requirements aim at enforcing properties related to COP at the implementation level. Second, we propose a revised and extended version of our component language called SCL [FDH08] to fulfill these requirements. The first major extension is a uniform solution to manage self-references. A second important extension is a solution that enforces encapsulation and communication integrity using an argument passing mechanism based on automatic bindings.

This article is organized as follows. Section 2 lists and motivates five requirements we identified for any COP infrastructure. We provide in Section 3 a study of the state of the art exhibiting limitations of representative COP approaches regarding the previous requirements. Section 4 describes basic features of the component programming language SCL as it was before this work. Next, Section 5 presents a set of new extensions to SCL and revisions of some of its core elements. We show how these extensions and revisions contribute to make the updated SCL fulfill our five requirements for COP. Then, we describe in Section 6 some experiments we made with the updated SCL. Finally, Section 7 concludes this paper by a summary and a presentation of some future work.

2. Requirements for Component-Oriented Programming

In this section, we present five requirements we identified for component-oriented programming. Each following subsection introduces and motivates one requirement to support COP by enforcing component-related properties at the implementation level.

2.1. *Decoupling*

To enable reuse, decoupling the components code one from the other is mandatory. The *decoupling* principle aims at avoiding hardwired references. Connections between components should be deferred until deployment or execution, as opposite to "traditional" OOP where connections between objects may be defined at design-time and hardwired inside constructors or initialization methods. Consider the example of a tcp server for networking of Figure 2. An initialization method in the TCPSERVER class directly references the ASYNCHRONOUSREQUESTPROCESSOR class to instantiate it. An instance variable of the class TCPSERVER stores the reference of the newly created object (instance of ASYNCHRONOUSREQUESTPROCESSOR). Such a direct reference to ASYNCHRONOUSREQUESTPROCESSOR in the source code of TCPSERVER is not desirable because it prevents reusing TCPSERVER with another request processor. COP should avoid such undesirable situation that hampers reuse. Components should never hold a reference on each other until they get assembled in a software.

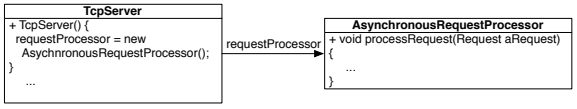

Figure 2: A coupled implementation of a tcp server in OOP

2.2. *Adaptability*

Adaptability is the ability to revise an assembly (a set of connected components) at run-time. For instance by

changing connections between components, adding new components, or removing existing ones. It eases developing context-aware applications that change according to their environment change [DL03, GBV08, GBV06].

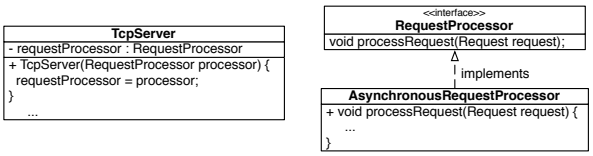

Figure 3: Constructor injection does not meet the adaptability requirement

Not all approaches support adaptation. Consider the example of Figure 3. The TCPSERVER relies on constructor injection [Fow04] to support the decoupling requirement. With this decoupling technique, adaptation is then impossible because the `requestProcessor` instance variable receives a value once for all in the constructor of TCPSERVER. Replacing the request processor of a tcp server after its creation is not possible when using constructor injection.

2.3. Unplanned Connections

Unplanned connections refer to connections between components developed in different contexts, or by different component developers. It is the application developer that decides and connects components. So, components that have compatible functionalities should be connectable, even-though their developers did not plan such connections. Indeed, it is impossible to a component developer to foresee all possible connections to components he develops. Predicting all connections is not possible that is why it should not be considered.

COP should support proper compositions by allowing application developers to find out compatible components. Therefore, a component should be self-documented. There should be a set of contracts [BJPW99] attached to every component to document its functionalities. Application developers can rely on these contracts to select components appropriate to the application's needs, and check what they are building.

Most COLs are Java-based and rely on names and types as discussed above. As shown on Figure 4, a connection between two components is only possible if there is an explicit sub-typing relation between their linked interfaces. This sub-typing relationship is mandatory even if different people developed these two components. Therefore, in this context, application developers should share some ontology and agree about names and types.

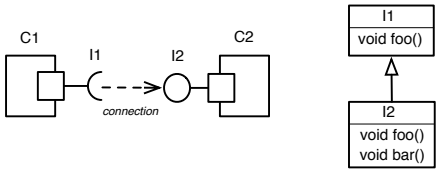

Figure 4: The problematic typing relation of independently developed components

This hypothesis is not realistic. Even if we assume the existence of standards. Component developers may choose different names or types for their components' functionalities. Still, two components might be compatible from the functionalities point of view. Consider simply two methods that have the same name, do the same processing, but have different parameter orders. They are syntactically different, though they are semantically equivalent. Therefore, a COL should provide facilities to application developers handling these situations and building unplanned connections between components that might be syntactically incompatible, though semantically compatible.

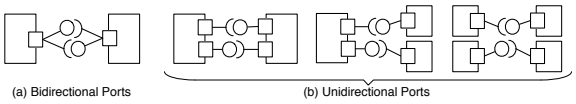

Figure 5: Unidirectional ports better support reuse by enabling unplanned connections

Another issue related to unplanned connections is the structure of ports. The UML component model support bi-directional ports. A bidirectional port can only be connected to another bi-directional compatible port as shown on Figure 5(a). This rule constraints the set of possible connections and goes against the unplanned connections principle. Indeed, bidirectional ports enforce the unicity of the connected component. On the other hand, unidirectional ports offer more reuse opportunities since they enable multiple possibilities for connections as shown in Figure 5(b).

2.4. Encapsulation and Communication Integrity

One of the properties of components is encapsulation. A component only knows the contracts of its connected components. Internal structure of a component is always hidden. Connections are the only means of interaction.

It is important to preserve encapsulation for two reasons. First, the internal structure of components may

change, possibly at run-time such as in the case of adaptive components [GBV08, GBV06, DL03]. Second, a component may want to enforce some extra-functional behavior (such as log or authentication) for some of its functionalities. In this context, *communication integrity* stands for ensuring that interactions among components preserve encapsulation.

The encapsulation problem also reveals an issue about *self-references*. How a component should reference itself when invoking services of other components (self as a parameter) or when answering some invocations sent by other components (self as a result)? How should it invoke its own services? We can draw here a parallel with work on "composition filters" [AWB+93] in which multiple layers can wrap an object and filter incoming or outgoing messages. There exist different pseudo-variables to reference the current object in its own computations. Developers make an explicit decision whether to short-cut the filters or not by choosing the appropriate pseudo-variable.

2.5. Uniformity

Most of the existing COLs and approaches to COP rely on object-oriented languages. Some, such as Fractal and EJB allow designers to reason upon components. But developers still have to deal with objects during development. Therefore, we end up with a gap between design and implementation that makes development and maintenance difficult as shown in Figure 6. Indeed, software engineers have to deal with different abstractions and map components to objects and keep both representations synchronized. Some component-based approaches rely on code generation to automatize this mapping. On the one side, it has the nice property of reducing the amount of code typed by developers. On the other side, it makes debugging difficult and reverse engineering almost impossible. Contrary to the use of an object-oriented language or code generation, a component-oriented language (COL) enables the programmer to directly manipulate component concepts (components, connections, ...) in the source code. A COL eases code writing and code generation from ADL specifications as well as reverse engineering because the concepts are close enough. Only COLs offer such a continuum between design, implementation and even runtime.

2.6. Summary

In summary, we claim that a COP approach should at least fulfill the five following requirements:

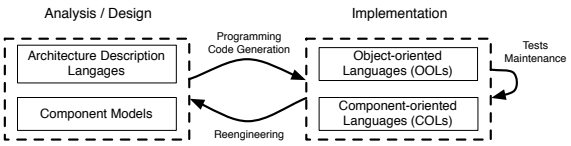

Figure 6: The need for Component-Oriented Languages to reduce the gap between design and implementation.

1. A component must not reference directly another external component. (R1 - Decoupling)
2. Connecting and disconnecting components must be possible at run-time. (R2 - Adaptability)
3. Connections between semantically compatible components must be possible even if they were not planned by developers. (R3 - Unplanned connections)
4. Component encapsulation must be ensured especially by enforcing communication integrity. (R4 - Encapsulation and Communication Integrity)
5. Only component-related concepts should be used throughout the software life-cycle from design to deployment, passing through implementation, and even runtime. (R5 - Uniformity)

In the next section we present a state of the art of the main component-based approaches regarding these five requirements.

3. State of the Art

This section is organized based on the requirements presented in Section 2. Each subsection discusses one requirement and how most representative of current COP practices address it. Contrary to general state of the art in the component field [LW05, CSVC10], this section focuses as much as possible on approaches at the implementation level (cf. Figure 6).

3.1. Decoupling

In approaches such as EJB, Fractal and ArchJava, decoupling relies on the *dependency injection* technique [Fow04]. Figure 7 illustrates this approach in the context of the Java programming language where an interface acts as a contract specification.

Only interfaces are used as types. So, the class TCPSERVER only references the interface REQUESTPROCESSOR (type of the `requestProcessor` instance variable and related methods). The developer

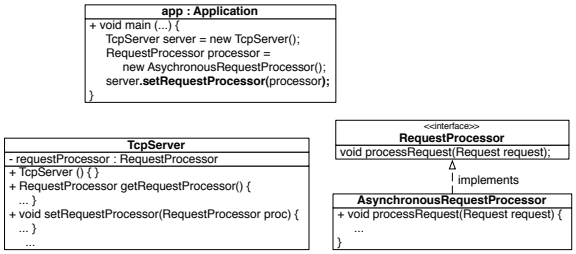

Figure 7: A decoupled implementation in OOP based on setter injection

of the application `app` may decide to use the AsynchronousRequestProcessor class and connects an instance of it to an instance of TcpServer. This solution also enables to dynamically change AsynchronousRequestProcessor by another implementation. Dependency injection mechanisms such as the setter injection one presented above are heavily used in component approaches such as in *Javabeans* and EJB. But, some other decoupling techniques such as constructor injection (discussed in Section 2.2), are not dynamically adaptable as we will see in the next section.

3.2. Adaptability

The Enterprise Javabeans (EJB) and the Corba Component Model (CCM) [Gro06] only support static connections and developers have to stop an application to change its architecture. ArchJava supports dynamically creating and connecting components. Nevertheless, the set of possible connections at run-time should be explicitly described by the application (or the composite component) developers.

Fractal does support adaptation by providing structural reflection [BCS02]. Therefore, developers can build components or applications that reason and act upon their structure and connections.

3.3. Unplanned Connections

In COP approaches such as EJB, Fractal and ArchJava, components provided and required functionalities are expressed using syntactic contracts [BJPW99]. That is, they rely on matching types to ensure the validity of connections between components. Figure 7 provides an example of such contract in an OOP context. Definition of class TCPServer states that it requires an object compliant with the interface RequestProcessor. Definition of class AsynchronousRequestProcessor states that every instance is compliant with the interface RequestProcessor. In this example, the contract is the compliance with the RequestProcessor interface.

3.4. Encapsulation and Communication Integrity

Ensuring communication integrity is a challenging issue. Consider Julia the Java-based implementation of the Fractal [BCL+06] component model. In Fractal, each component has a content that holds its internal state and implements its behavior. The component's content is wrapped by an envelope consisting of interfaces that are the hooks for connections from other components. Each Fractal component materializes in the Julia OO framework, as a set of objects. Each interface is represented by an object. The component content is represented by an object too[3].

The top part of Figure 8 shows an example of two Fractal components (C1 and C2) connected thanks to a binding between their respective client interface IC1 and server interface IS2. The bottom part of Figure 8 shows the object counterparts of components C1, C2 in Julia. Through the established connection, $C1_{content}$ is able to make a service invocation through IC1 and IS2 that will be eventually executed by $C2_{content}$. Assume that component C2 provides service `setX` through its interface IS2. $C2_{content}$ may store the argument it receives during the execution of its `setX` service. In this example, the argument is a reference to the $C1_{content}$ object (`this` has been passed). $C2_{content}$ may use this reference to directly communicate with the $C1_{content}$ object later in the program execution. We face here a violation of the communication integrity since it "short-cuts" the component interfaces.

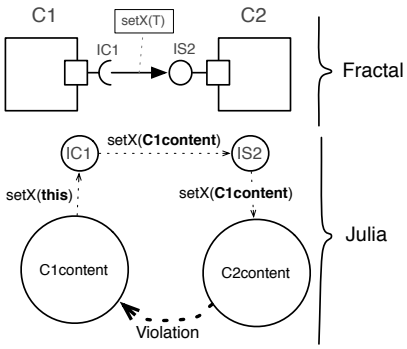

Figure 8: Violation of the communication integrity in Julia [LCL06]

In [LCL06], they identified this issue and solved it in Julia by supervision mechanisms at runtime. But it brings a runtime overhead to check the validity of communications. Archjava uses another solution based on a

[3]This is true for the primitive components. The content of composite components is a component assembly. Our analysis still hold for composites but we do not discuss this.

specific type system named *AliasJava* [Ald03, ACN02]. This solution ensures that components can communicate only through connections declared by developers and statically checked. Developers put annotations on variables to express precisely the expected behavior. With these annotations, it's possible in ArchJava to express properties such as uniqueness (an object can only be referenced by a unique variable in the system) or sharing (i.e. an object is shared among different components). But this solution then requires static analysis and validation of the data flow which restricts the run-time possibilities. A better solution is to add a mechanism that prevents communication violations by construction to avoid checking it.

3.5. Uniformity

ArchJava is a step forward since it is a COL and it bridges the gap between design and implementation. But it does not address the uniformity issue. Indeed, ArchJava provides concepts related to both OOP and COP. The following example shows the code of two component classes: `Helloer` and `StdInputOutput`. An instance of `Helloer` named *h* is then connected to *p* an instance of `StdInputOuput` through their respective ports *printing* and *stdout*.

```
component class Helloer {
   public port hello {
      provides String sayHello(String name);
   }

   public port printing {
      requires void print(String s);
   }

   String sayHello(String name) {
      printing.print("Hello, " + name);
   }
}

component class StdInputOutput {
   public port stdout {
      provides void print(String s);
   }
   void print(String s) {...}
}

Helloer h = new Helloer();
StdInputOuput p = new StdInputOutput();
connect h.printing, p.stdout;
h.hello.sayhello("luc");
```

The main drawbacks of ArchJava come from two non-uniformities:

Family	Approach	*R1 (Decoupl.)*	*R2 (Adapt.)*	*R3 (Unplanned con.)*	*R4 (Enc.&Com. Integrity)*	*R5 (Uniformity)*
OOL	*Javabeans*	+	+	-	-	-
Framework	Julia	+	+	-	-	-
	EJB 1&2	-	-	-	-	-
Code Generation	EJB 3	+	-	-	-	-
	CCM	+	-	-	-	-
	Sofa	+	+	-	-	-
COL	ArchJava	+	-	-	-	-
	ComponentJ	+	+	-	-	+
	CLIC	+	+	-	-	+

Table 1: Evaluation of main COP approaches classified by families for the five identified requirements.

1. A component can be connected, but it cannot be passed as a parameter
2. An object can be passed as a parameter, but it cannot be connected

This strict separation between components and objects is a difficulty for developers that must choose at design time if a concept should be implemented as a component or as an object. This decision has a deep impact on design and programming, and it is then difficult to change it in future software evolutions.

3.6. Summary

Table 1 shows a summary of the evaluation[4] of these five requirements for selected COP approaches classified in four families (as described in the introduction):

- **Using an object-oriented language (OOL).** One of the most powerful component model in this family is *Javabeans*. *Javabeans* satisfies R1 and R2 because its components connection mechanism relies on the Observer design pattern [GHJV95]. But it also uses standard OOP and syntactic conventions that do not fulfill R4 and R5. R3 is also not

[4] + means that an approach satisfies a requirement and – means that it does not.

fulfilled because a *Javabean* only supports event-based connections. It leads to a limited expressiveness compared to other component models.

- **Extending an OO framework.** Julia, the implementation of Fractal, is a popular representative of this family. We already discussed the drawbacks of Julia. Another member of this family is EJB. The first versions of EJB do not enforce decoupling since a client should reference its provider.

- **Using code generation.** All approaches based on code generation cannot fulfill R5. They also have a limited support for R2 and R3 because adaptation often implies to regenerate the code at runtime. EJB 3 supports decoupling thanks to dependency injection and Java annotations. Sofa [PBJ98, BHP06] and CCM are language independent component models. Component interfaces are described in an abstract Interface Description Language (IDL) and then used to eventually generate partial implementation in some programming language such as Java or C++. Sofa also supports component updating at runtime so it is possible to replace a composite with another one with a different internal architecture.

- **Using a component-oriented language (COL).** This family is the most promising regarding R3, R4 and R5. Unfortunately there are few COLs. ArchJava does not support R3 and R5. It only proposes a static solution for R4. ComponentJ [SC00] is a COL built as a Java extension and CLIC [BF09] is a symbiotic component model with the Smalltalk object model. They both give a uniform view to their developers (R5) but they also do not fulfill the requirements R3 and R4.

This focused state of the art shows that no component-based approach in the literature fulfills all of the five requirements we identified.

Section 4 presents SCL that already partially addressed some of these requirements such as CLIC. Then, Section 5 presents some extensions to SCL that complete it and make it fulfills these five requirements.

4. Presentation of SCL

The design of SCL has originally [FDH08] been driven by the aim of building a clean COL in an incremental way by adding only features that enable COP. This section describes some of the core elements of SCL.

4.1. Structure of a Component

A component is an instance of a descriptor which is similar to the concept of class in OOP. Each component has a set of ports which are the only means to interact with other components. Component services (operations) can be invoked through its provided ports. Symmetrically, it can send service invocations to other components through its required ports.

Figure 9 shows the declaration of a TCPServer component descriptor in SCL[5]. A TCPServer has two provided ports (*RequestHandling* and *LifeCycle*) and one required port (*RequestProcessing*). Each port is described by an interface i.e. the set of available services through this port, described by their signatures.

```
(Component newDescriptor: #TCPServer)
   providedPorts: {
      #RequestHandling->#(handle:).
      #LifeCycle->#(start stop) };
   requiredPorts: {#RequestProcessing->#(process:)}.

(TCPServer new @ #RequestProcessing) bindTo:
        (AsynchronousRequestProcessor new @ #Processing)
```

Figure 9: Example of a TCPServer component descriptor and its instantiation and binding.

The `new` primitive of SCL creates a component from a descriptor. It returns a reference to a special provided port of the newly created component. Indeed, in SCL, ports are the only means to handle components. There exist no other means to reference a component.

SCL supports *multi-port* to deal with multiple related connections. A *multi-port* is an indexed collection of ports (all required or all provided). *multi-port* enable to dynamically and automatically manage grouped ports of the same kind (required or provided). Indeed, a multi-port is a collection with unlimited number of ports. New ports can be added to the collection upon need. The following piece of code propose a revised version of the TCPServer that has a multi-port to maintain connections to multiple request processors; each one dealing with one request at a time.

```
(Component newDescriptor: #TCPServer)
   providedPorts: {
      #RequestHandling->#(handle:).
      #LifeCycle->#(start stop) };
   multiRequiredPorts:{#RequestProcessors->#(process:)}.
```

Ports are dedicated to support connections and service invocations. Connection validation can be checked using interfaces. Indeed, an interface corresponds to the

[5] SCL uses a Smalltalk compliant syntax.

contract of a port. In SCL, we only focus on syntactic contracts [BJPW99]. So, an SCL interface specifies signatures of services of a given port.

A connection between two ports is valid if their interfaces match. This matching is based on the inclusion relationship between the sets of service signatures. The use of unidirectional ports and the set inclusion relation for interface compatibility contribute to enable unplanned connections (cf. Section 2.3) in SCL.

4.2. Bindings and Connectors

As stated above, components can only be handled through their ports and components assembling consists in connecting their ports. Connections can be achieved either through *bindings* or through *connectors*.

A binding is a directed link from a *source* port to a *target* port. Service invocations that reach the source port are routed to the target port. The source and the target of a binding must be compatible ports i.e. their interfaces match as described in Section 4.1.

A port can be the target of multiple bindings. But it can be the source of only a single binding. Given a port `p` which is the source of an existing binding b_1, if we attempt to make `p` the source of another binding b_2, an exception is raised. The developer should first explicitly unbind `p` before using it as source for b_2.

One of the particularities of SCL is that the source and the target ports can be of any kind. Indeed, ports linked by a binding can be both required, or both provided. Alternatively, the source of a binding can be a required port and the target can be a provided port. Last, the source of a binding can be a provided port and the target can be a required port. The possibility to bind ports of the same kind in SCL allows developers to control and route service invocations along binding chains. A binding chain starts from a source port and ends by a provided port which belongs to a component that will execute the service corresponding to the invocation.

The last two lines in Figure 9 show the SCL code to establish a binding. It links a required port (*RequestProcessing*) to a provided one (*Processing*) of two newly instantiated components. This binding is the simplest possible form. It ensures that all service invocations that reach the *RequestProcessing* port will be redirected to the *Processing* port. This binding is only possible if the interface of `Processing` contains at least a service named `handle:`.

Bindings correspond to simple communication routes between two components. Developers can express more complex interactions involving two or more components with the concept of connector. A connector is a component dedicated to route and adapt service invocations from components which emit them to components which process them. A typical use of connectors is for connecting ports with mismatching interfaces. It plays the role of an adaptor and converts invocations to fit the target port interface. A connector should often be handwritten but it may also be partially generated.

4.3. Service Invocations

Service invocations issued by a component go out through a required port. The invocation is transmitted if the required port is the source of a binding. A successfully established binding implies that the target port has a compliant interface with source port interface. So, only the following cases can happen:

- the target port is a source of a second binding. So, the invocation is forwarded.
- the target port is not a source of any binding and it is a provided port. So, the service corresponding to the invocation is looked up in its associated component and executed.
- otherwise, the invocation fails raising an exception.

4.4. Composites

Composites are components that encapsulate other components often called subcomponents. Composites are easily supported in SCL thanks to a visibility property associated to ports. Ports are either *external* i.e. accessible from outside the code of a component or *internal* i.e. only accessible from the implementation of the component. Figure 10 shows an example of composite instance of TCPServer. This example shows that internal ports are similar to instance variables in OOP. They are encapsulated inside the component and can only be accessed by the component's implementation. The example also shows that the subcomponent is an instance of SmallInteger. SCL offers a uniform component-based view; even primitive types can be connected. Finally, it also worth noting that in SCL, an assignment is just a port binding.

4.5. Evaluation

Table 2 shows the evaluation of SCL regarding the five requirements we identified in Section 2 for COP.

On the one hand, SCL fully satisfies R1, R2 and R5 because:

R1 (Decoupling). In SCL a component description only references declared ports. Contracts of ports are expressed as interfaces.

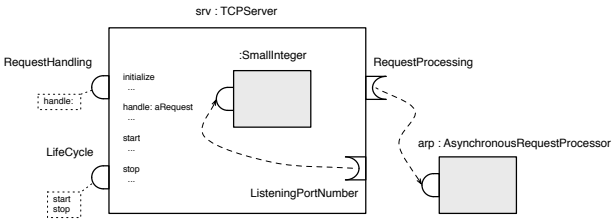

Figure 10: Example of composite with external and internal ports.

Requirements	SCL
R1 (Decoupling)	+
R2 (Adaptability)	+
R3 (Unplanned connections)	−
R4 (Encapsulation and Communication Integrity)	−
R5 (Uniformity)	+

Table 2: Requirements evaluation for SCL

R2 (Adaptability). It is possible to dynamically bind and unbind ports in SCL. Furthermore, it is possible to dynamically create connectors and connect multiple components if they have compatible interfaces.

R5 (Uniformity). SCL is a COL where only component related concepts where introduced. Main concepts are: port, binding, connector and component. Moreover every entity is considered as a component including basic entities such as numbers.

On the other hand, SCL does not fulfill R3, R4 because:

R3 (Unplanned connections). Adding a component as a new participant in a connection at run-time implies dynamically creating a binding. But establishing such a binding at runtime still requires that ports interfaces match. Therefore this connection should be planned at design time.

R4 (Encapsulation and Communication Integrity). As in Fractal (cf. Section 3.4), it is possible in this version of SCL to break the integrity of communication passing an internal port as an argument.

5. Extending SCL to fulfill COP requirements

We present in this section a set of SCL extensions and revisions to make it satisfy all the five requirements we identified for COP.

5.1. Optional Interfaces and High-Level Connectors

Previously, a port had to be described by one interface. This constraint restricts the adaptability of the connectors at runtime because each connector's port should be statically described by an interface. It also forbids unplanned connections because it requires interface conformance between bound ports.

To better support R3, we enable having ports without interface. Now, a port accepts bindings to any other port, regardless of the interface of the latter. It is typically useful to write and reuse high-level connectors without specifying the sources and targets interfaces as shown in Figure 11. All connectors are now of the same high-level form. They all have two multi-ports. The *sources* multi-port contains provided ports through which the connector receives invocations. A connector intercepts incoming invocations thanks to *glue services* (defined hereafter). These special services adapt invocations and transmit them through the *targets* multi-port.

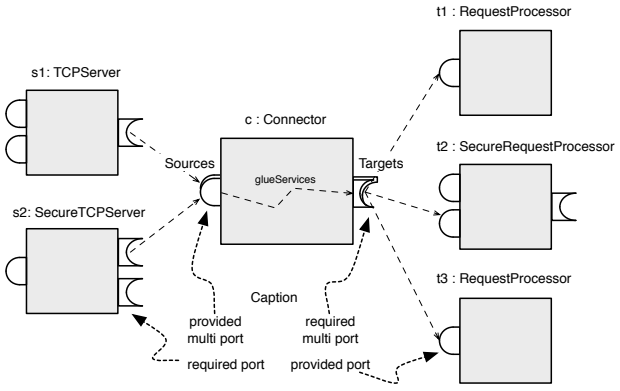

Figure 11: General Form of a Connector

With optional interface, interface conformance is not enforced anymore. So, a port may receive any service invocations. We introduce *glue services* to process unplanned service invocations. A glue service is analogous to the `doesNotUnderstand:` method in Smalltalk but, it is specific to a single provided port. The glue service associated to a port is automatically executed each time an invocation is received through this port and the component does not define the corresponding service. A component is therefore able to deal with service invocations it receives even though it does not have a directly matching implementation.

5.2. Service Invocations

Service invocations issued by a component still go out through a required port. The invocation is transmitted if the required port is the source of a binding. Otherwise, the invocation fails raising an exception.

For provided ports, the invocation handling is now more complex. Four prioritized rules determine how the invocation is processed.

1. The rule of highest priority corresponds to the situation where the provided port is the source of a binding. In this case, the invocation is routed to the port that is the target of the binding.

2. The pre-condition of second rule is checked if the provided port that receives the invocation is not the source of any binding. The second rule is applicable if the component that owns the provided port implements a service that matches the invocation. This matching is based on the service name and the number of its arguments.

3. Then the third rule pre-condition is checked if none of the above rules apply. The third rule is applicable if the component that owns the provided port implements a *glue service* for this port.

4. The rule of lowest priority consists in raising an exception to signal invocation failure. It is performed if none of the three previous rules apply.

5.3. Internal Ports and Self-References

In SCL, every service invocations is achieved through a port. In our former work on SCL, it was possible to violate the encapsulation by passing an internal port as argument or a result for an invocation.

We extended SCL to improve encapsulation and communication integrity (R4) by introducing an extra constraint on service invocation. Only external ports can be passed as arguments of service invocations that go outside a component. Therefore, references to internal ports can never be accessed by external components[6]. This applies to the special internal port *Self*.

Self is an internal port available in every component. It allows the component to invoke its own services even if they are not available to the outside through any external port. Through *Self*, a component provides all services it implements. Figure 12 shows that the internal `kill` service can be invoked through the *Self* port.

[6]Internal ports can still be connected to ports of sub-components of a composite.

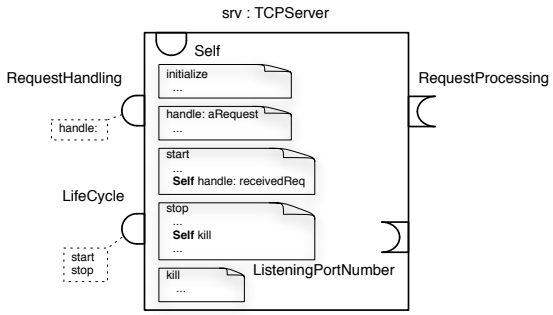

Figure 12: Some uses example of *Self* port.

Likewise other ports, the *Self* port can be bound (i.e. be the source of a binding) or can even have a glue service attached to it. Figure 13 illustrates these two situations by showing the code of the initialize service of two composite components. In Composite1, *Self* is bound and all invocations made through *Self* in the Composite1 code (such as in the *doit* service invocation made in the *foo* service) will be treated by the subcomponent bound to the internal port *Sc1*. In Composite2, a glue service is attached to *Self*. In this example, if Composite2 implements a matching service, it will be executed, otherwise the invocation is delegated to the subcomponent.

These examples show that the developer is able to control service invocation flow with bindings. Using connectors, even more complex examples can be set up.

5.4. Argument Passing by Connection

In Section 5.3, we introduced a solution to avoid exposing the internals (internal ports and subcomponents) of a component to other components. We modified the SCL service invocation to prevent connections from internal ports to external components received as parameters or invocation result. This modification completes our encapsulation enforcement and ensures communication integrity.

Our extension relies on the connection mechanism. Every component is equipped with a multi-port of required ports called "Args" to which the service arguments will be automatically bound before the execution of a service. The names used in the code for arguments are transparently aliased to *Args* ports. All *Args* ports are unbound at the end of a service execution.

Figure 14 shows an example. Initially, the TCPServer component is executing its `handle:` service. The argument named aRequest is an alias to the first port in its multi-port *Args*. The first step shows that

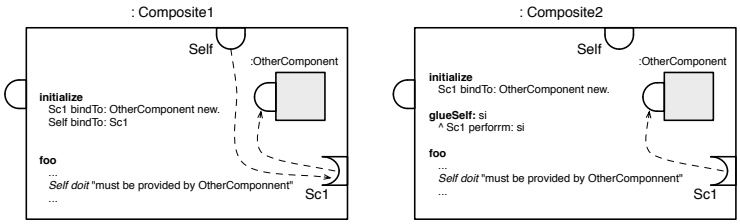

Figure 13: *Self* port can be delegated or even attached to a glue service.

this TCPServer invokes a `process:` service through its required port *RequestProcessing* passing the aRequest as an argument. The second step shows that invocation is transmitted through the binding of the *RequestProcessing* port. The third step applies before executing the `process:` service implemented by the ASynchronousRequestProcessor component. All arguments passed in the service invocation are bound[7] to the *Args* ports of this component. The service is then executed in the fourth step before unbinding the *Args* ports.

This argument passing mechanism is uniform with the SCL model. It prevent programmers to store references to third party ports because they only manipulate *Args* ports that belong to the current component.

The SCL interpreter deals with situations such as recursive invocations and concurrency by managing a collection of argument bindings. When a component receives an invocation of a service before returning the result of previous ones. Upon the reception of a new invocation, the interpreter stores the *Args* bindings, next it unbinds the *Args* port, then it binds it with the new arguments before performing the the latest invocation. The interpreter restores the bindings of *Args* of the previous service invocation once the latest invocation returns. A similar solution enables dealing with concurrency.

5.5. Summary

We introduce multiple enhancements and extensions in this revised version of SCL such as: optional interfaces, service invocation handling, and argument passing by connection to name a few. With these new features, SCL now fulfills R3 and R4.

R3 (Unplanned connections). Connecting components semantically compatible but with different interfaces is now easier with high-level connectors thanks to optional interfaces. A port without interfaces accepts bindings to other ports with any interface. Last, ports may be provided with glue services, those are services that are called when no matching implementation is found for an incoming service invocation.

R4 (Encapsulation and Communication Integrity). Two major features in SCL enforce this requirement. On the one hand, SCL forbids passing internal ports as invocation arguments, thus avoiding connections from the outside to the internals of a component. On the other hand, invocation arguments are referenced through a multi-port (the `Args` collection of ports) which is unbound after the invocation returns. Therefore, SCL prevents connecting a component's internal ports to external components.

6. Prototype and Experimentation

SCL has been initially prototyped in Squeak [IKM+97] which is an open-source Smalltalk implementation. Thanks to the powerful meta-object protocol of Smalltalk [Duc99, BD06], writing and extending the SCL prototype has been much simpler than writing a parser and a complete interpreter. The main drawback of this implementation is on the performance side. But our aim with this prototype is to demonstrate that it is possible to implement a COL that fulfills all the requirements we identified.

Figure 15 shows the general architecture of the extended SCL prototype [scl]. This architecture has four layers.

1. The top most layer contains the Smalltalk kernel classes (OBJECT and OBJECT CLASS) that have been extended.

2. The second layer contains the classes of the implemented SCL model. It would have been better

[7] Arguments are always ports in SCL.

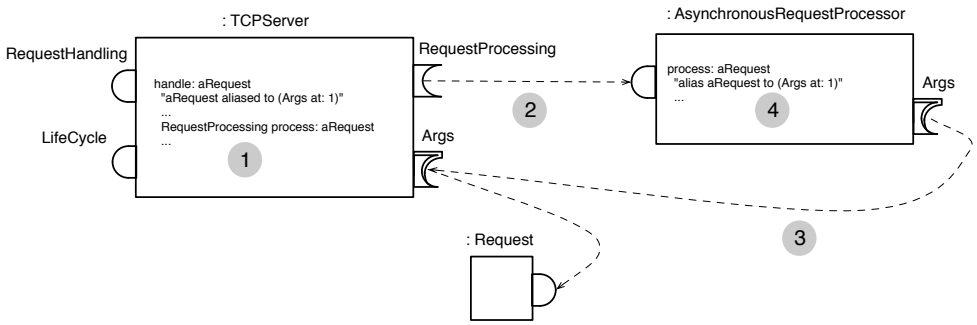

Figure 14: The four steps of arguments passing by automatic connection.

to implement SCL component descriptors by directly inheriting from OBJECT. However if component descriptors were not classes they would have been incompatible with standard Smalltalk tools (class browser, versioning tool, ...). That is why component descriptors (COMPONENT CLASS in Figure 15) have been implemented by subclassing OBJECT CLASS. It is at this level where we introduced the new features presented in Section 5.

3. The third layer contains all available SCL component descriptors that are automatically implemented as subclasses of COMPONENT.

4. The fourth layer contains components instances of descriptors.

SCL developers only deal with SCL descriptors and components present in layers 3 and 4. They never work at the level of layers 1 and 2.

Bindings and Service invocations. Service invocations in SCL use the same syntax as message sending in Smalltalk. But in SCL, the syntactic receiver of a service invocation is always a port. Port objects in the implementation can then intercept all service invocations for their components thanks to the `doesNotUndestand:` Smalltalk method. Port objects implement the routing algorithms of service invocations described previously and it is also possible to automatically bind and unbind arguments.

The TCPServer example. We implemented the TCPServer as described previously. Figure 16 shows the declaration of three components. The component TcpServerApp describes an application and connect a TcpServer to a RequestProcessor. This example shows the binding of two ports with mismatching interfaces. Indeed, TcpServer invokes a service named `handle:` through its port *RequestHandler*. But RequestProcessor provides a service named `process:` through its port *RequestProcessing*. This example uses `bindTo:withAliases:` to bind and easily adapt mismatching interfaces of the ports. It is just syntactic sugar to not write a specific BinaryConnector.

```
(Component newDescriptor: #TcpServer
   category: 'Scl-Examples-TCPServer')
   requiredPorts: #(#RequestHandler);
   providedPorts: {
      #RequestHandling->#(#handle:).
      #Lifecycle->#(#start #stop)}.

TcpServer>>handle: aRequest
   RequestHandler handle: aRequest

(Component newDescriptor: #RequestProcessor
   category: 'Scl-Examples-TCPServer')
   requiredPorts: {};
   providedPorts: {#RequestProcessing->#(#process:)}.

(Component newDescriptor: #TcpServerApp
   category: 'Scl-Examples-TCPServer')
   requiredPorts: {};
   providedPorts: {#RequestHandler->#(#handle:);
   internalRequiredPorts: #(#pServer #pProcessor);
   internalProvidedPorts: {}}.

TcpServerApp>>init
   pServer bindTo: TcpServer new.
   pProcessor bindTo: SclRequestProcessor new.
   pServer @ #RequestHandler
      bindTo: pProcessor @ #RequestProcessing
      withAliases: { #handle: -> #process: }.

TcpServerApp new start
```

Figure 16: The TCPServer example in SCL

Requirements discussion. This example shows that the declared components RequestProcessor and TCPServer are decoupled (Requirement R1). The only admitted couplings are between a composite (e.g. TcpServerApp) and its sub-components (e.g. Re-

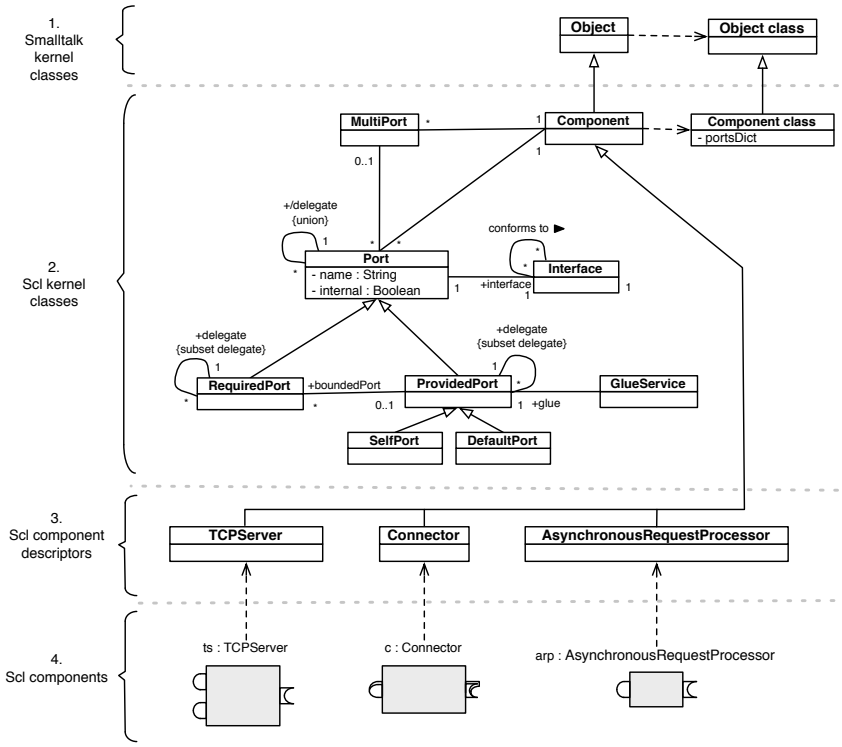

Figure 15: Overview of the extended SCL prototype

questProcessor and TCPServer). This makes sense because the developer of a composite makes fixed decisions about internal details. These details are encapsulated to third parties that are only able to access external ports of the composite. This example also illustrates requirements R2 and R3. First, the connection between TcpServer and RequestProcessor is unplanned (Requirement R3). Indeed, their connected ports have mismatching interfaces. Second, the connector is built at runtime. And it can also be adapted later during the program execution (Requirement R2). Besides, SCL ensures that the connected components will be able to share data (`requests`) without violating communication integrity (Requirement R4). Finally, this example illustrates that only components-related concepts are used. Therefore, SCL also fulfills the uniformity requirement R5.

7. Conclusion and Perspectives

The contribution of this paper is twofold. First, we identified five requirements to fully support Component-Oriented Programming (COP): decoupling, adaptability, unplanned connections, encapsulation and uniformity. The study of the state of the art shows that no existing work addresses all these requirements. Unplanned connections is often an issue, since most approaches rely on typing to check components compatibility. Therefore, component developers must know about types used in other components to enable direct connections. Another major issue is a lack of communication integrity. Components should be able to interact while preserving their encapsulation. Last, non-uniformity is a frequent criticism of existing work. We often find COP concepts superposed with the OO ones either at the implementation-level or even at the model-level. We advocate that only COP concepts should be used from design to implementation.

The second contribution of this article is a Component-Oriented Language (COL) that satisfies requirements mentioned above. We started from our previous work called SCL which we extended to make it fully compliant with COP requirements. SCL was first thought as a uniform language for COP. It thus satisfies the uniformity requirement. We show that the extended SCL also satisfies the other requirements. An important evolution of SCL results in enforcing encapsulation and communication integrity. It ensures that no

connection can be set from/to the internals of a component to/from external components. This is achieved by forbidding outgoing service invocations that reference internal ports and by aliasing parameters of incoming service invocations. Aliasing consists in referencing parameters through a dedicated external port on every component.

Regarding future work, we plan to study the merge of SCL with our other work CLIC [BF09]. The idea is to enable component-based programming while still taking advantage of Smalltalk tools and libraries. One possible direction would be to use Helvetia [RGN10] to embed a component language in Smalltalk without breaking tools.

Bibliography

[ACN02] Jonathan Aldrich, Craig Chambers, and David Notkin. Architectural Reasoning in ArchJava. In *ECOOP '02: Proceedings of the 16th European Conference on Object-Oriented Programming*, pages 334–367, London, UK, 2002. Springer-Verlag.

[Ald03] Jonathan Aldrich. *Using Types to Enforce Architectural Structure*. PhD thesis, University of Washington, August 2003.

[AWB+93] Mehmet Aksit, Ken Wakita, Jan Bosch, Lodewijk Bergmans, and Akinori Yonezawa. Abstracting object interactions using composition filters. In Rachid Guerraoui, Oscar Nierstrasz, and Michel Riveill, editors, *ECOOP Workshop*, volume 791 of *Lecture Notes in Computer Science*, pages 152–184. Springer, 1993.

[BCL+06] Eric Bruneton, Thierry Coupaye, Matthieu Leclercq, Vivien Quéma, and Jean-Bernard Stefani. The fractal component model and its support in java: Experiences with auto-adaptive and reconfigurable systems. *Softw. Pract. Exper.*, 36(11-12):1257–1284, 2006.

[BCS02] E. Bruneton, T. Coupaye, and J.B. Stefani. The Fractal Composition Framework. Technical report, The Object Web Consortium, July 2002.

[BD06] Alexandre Bergel and Marcus Denker. Prototyping languages, related constructs and tools with squeak. In *In In Proceedings of the Workshop on Revival of Dynamic Languages (co-located with ECOOP'06*, 2006.

[BF09] Noury Bouraqadi and Luc Fabresse. Clic: a component model symbiotic with smalltalk. In *IWST '09: Proceedings of the International Workshop on Smalltalk Technologies*, pages 114–119, New York, NY, USA, 2009. ACM.

[BHP06] Tomas Bures, Petr Hnetynka, and Frantisek Plasil. Sofa 2.0: Balancing advanced features in a hierarchical component model. In *SERA '06: Proceedings of the Fourth International Conference on Software Engineering Research, Management and Applications*, pages 40–48, Washington, DC, USA, 2006. IEEE Computer Society.

[BJPW99] Antoine Beugnard, Jean-Marc Jézéquel, Noël Plouzeau, and Damien Watkins. Making Components Contract Aware. *Computer*, 32(7):38–45, 1999.

[BMH06] Bill Burke and Richard Monson-Haefel. *Enterprise JavaBeans 3.0*. O'Reilly, Beijing, 5. edition, 2006.

[CSVC10] Ivica Crnković;, Severine Sentilles, Aneta Vulgarakis, and Michel R.V. Chaudron. A classification framework for software component models. *IEEE Transactions on Software Engineering*, 99(PrePrints), 2010.

[DL03] Pierre-Charles David and Thomas Ledoux. Towards a framework for self-adaptive component-based applications. In Jean-Bernard Stefani, Isabelle M. Demeure, and Daniel Hagimont, editors, *DAIS*, volume 2893 of *Lecture Notes in Computer Science*, pages 1–14. Springer, 2003.

[Duc99] Stephane Ducasse. Evaluating message passing control techniques in smalltalk. *Journal of Object-Oriented Programming (JOOP*, 12:39–44, 1999.

[FDH08] Luc Fabresse, Christophe Dony, and Marianne Huchard. Foundations of a simple and unified component-oriented language. *Comput. Lang. Syst. Struct.*, 34(2-3):130–149, 2008.

[Fow04] M. Fowler. Inversion of control containers and the dependency injection pattern., 2004.

[GBV06] Guillaume Grondin, Noury Bouraqadi, and Laurent Vercouter. Madcar: an abstract model for dynamic and automatic (re-)assembling of component-based applications. In *The 9th International SIGSOFT Symposium on Component-Based Software Engineering (CBSE 2006)*, June 29th – 1st July 2006.

[GBV08] Guillaume Grondin, Noury Bouraqadi, and Laurent Vercouter. Component reassembling and state transfer in madcar-based self-adaptive software. In Bertrand Meryer Richard F. Paige, editor, *Objects, Components, Models and Patterns, Proceedings of TOOLS Europe 2008*, LNBIP, pages 258–277, Zurich, Switzerland, June/July 2008. Springer.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns : Elements of Reusable Object-Oriented Software*. Addison Wesley, March 1995.

[Gro06] Object Management Group. CORBA Component Model 4.0 Specification. Specification Version 4.0, Object Management Group, April 2006.

[Ham97] Graham Hamilton. JavaBeans. API Specification, Sun Microsystems, July 1997. Version 1.01.

[IKM+97] Dan Ingalls, Ted Kaehler, John Maloney, Scott Wallace, and Alan Kay. Back to the future: the story of Squeak, a practical Smalltalk written in itself. In *OOPSLA '97: Proceedings of the 12th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 318–326, New York, NY, USA, 1997. ACM Press.

[LCL06] Marc Léger, T. Coupaye, and Thomas Ledoux. Contrôle dynamique de l'intégrité des communications dans les architectures à composants. In S. Vauttier R. Rousseau, C. Urtado, editor, *Langages et Modèles à Objets*, pages 21–36. Hermès-Lavoisier, 2006.

[LW05] Kung-Kiu Lau and Zheng Wang. A taxonomy of software component models. In *EUROMICRO-SEAA*, pages 88–95. IEEE Computer Society, 2005.

[McI68] M. D. McIlroy. Mass produced software components. In P. Naur and B. Randell, editors, *Proceedings, NATO Conference on Software Engineering*, Garmisch, Germany, October 1968.

[MH99] Richard Monson-Haefel. *Enterprise JavaBeans*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 1999.

[MT00] Nenad Medvidovic and Richard N. Taylor. A Classification and Comparison Framework for Software Architecture Description Languages. *Software Engineering*, 26(1):70–93, 2000.

[Ous05] M. Oussalah. *Ingénierie des composants : concepts, techniques et outils*. Editions Vuibert, 2005.

[PBJ98] F. Plásil, D. Bálek, and R. Janecek. SOFA/DCUP: Architecture for component trading and dynamic updating. In *CDS '98: Proceedings of the International Conference on Configurable Distributed Systems*, page 43, Washington, DC, USA, 1998. IEEE Computer Society.

[RGN10] Lukas Renggli, Tudor Gîrba, and Oscar Nierstrasz. Embedding languages without breaking tools. In *In ECOOP 2010: Proceedings of the 24th European Conference on Object-Oriented Programming*, 2010.

[SC00] João Costa Seco and Luís Caires. A basic model of typed components. *Lecture Notes in Computer Science*, 1850:108–129, 2000.

[scl] Simple component language prototype. http://www.squeaksource.com/Scl.

[SDK+95] Mary Shaw, Robert DeLine, Daniel V. Klein, Theodore L. Ross, David M. Young, and Gregory Zelesnik. Abstractions for Software Architecture and Tools to Support Them. *Software Engineering*, 21(4):314–335, 1995.

[Szy02] C. Szyperski. *Component Software: Beyond Object-Oriented Programming (2nd Edition)*. Addison-Wesley, 2002.