Component-Oriented Programming: From Requirements to Language Support

L. Fabresse[1,*], N. Bouraqadi[1], C. Dony[2], M. Huchard[2]

Abstract

Although component-based software development have been subject to extensive research for many years, most software systems are still based on the object-oriented paradigm. We believe that one of the main causes is a lack of support for Component-Oriented Programming (COP). Indeed, a lot of work proposed component models such as Unified Modeling Language (UML), Enterprise JavaBeans (EJB) or Corba Component Model (CCM) that are only available at design time. At the implementation stage, object-oriented languages are mainly used which prevent developers to fully switch to COP. In this paper, we identify five requirements (decoupling, adaptability, unplanned connections, encapsulation and uniformity) for COP based on an analysis of the state of the art and limitations of existing work. We then propose an extended version of the Scl component language that fulfills these requirements.

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

1. Introduction

For decades, component-based software development (CBSD) promises better reusability of software pieces called *components* [McI68, Szy02, LW05, CSVC10]. Past researches have introduced or adapted concepts such as *component*, *port*, *architecture*, *composite* and mechanisms such as *connection* or *composition*. However, CBSD is still not widely used in practice. One of the reasons is because there are few solutions that really enable Component-Oriented Programming (COP). COP is twofold as shown in 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].

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

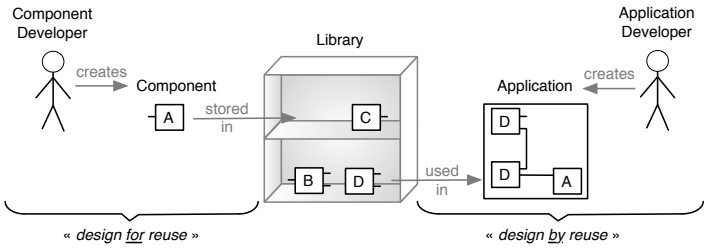

Figure 1: The Component-Oriented Programming Duality.

There are four families of COP approaches:

1. Using a general 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. Doing some model transformation such as generating OO code from a component specification (such as annotations in EJB3.0 [BMH06])
4. Using a component-oriented language (COL) (such as ArchJava [ACN02], ComponentJ [SC00])

The study of COP approaches from these different families allows us to exhibit some open issues in current COP practices. The contributions of this article are multiple. First, we propose a list of requirements for COP based on the open issues detected in the state of the art. These requirements will make COP approaches more usable at the implementation level. Second, we propose a revised and extended version of our component language called SCL [FDH08] to fulfill these requirements. A first extension proposes a uniform solution to manage self-references in a COL. And a second one solves the encapsulation and communication integrity issue using an argument passing mechanism based on automatic bindings.

This article is organized as follows. Section 2 is a study of the state of the art exhibiting open issues in some representative COP approaches. This section ends with a list of requirements for COP. Section 3 describes the revised version of some core elements of SCL which are descriptors, components, ports, interfaces, bindings, connectors and composites. Section 3.5 and Section 3.6 present the two main extensions of SCL. Section 3.5 is about allowing self references in a COL while ensuring the encapsulation and the uniformity requirements. Section 3.6 describes connection-based argument passing which ensures encapsulation and communication integrity in SCL. Finally, Section 4 concludes this paper by a summary and presents some future work.

2. Component-Oriented Programming: Open Issues and Requirements

In this section, we present five open issues we identified in today component-oriented programming practices. Our study is focused on Julia and ArchJava because they both enable COP at implementation level and on some high level approaches such as Unified Modeling Language [Obj10] (UML), Corba Component Model [Gro06] (CCM) or Enterprise Javabeans [BMH06] (EJB). We believe that these approaches are representative of current COP practices. From these open issues, we propose some requirements to support COP at the implementation level.

2.1. Decoupling

In order to enable reuse, components should be decoupled one from the other. 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. An initialization method in some class A may directly reference another class B in order to instantiate it for example. This is generally used for composition and aggregation, and the reference of the newly created object (instance of B) is stored in an instance variable of the class A. Such a direct reference to B in the source code of A is not desirable because it prevents to reuse A with another B-compliant implementation. Components should never hold a reference on each other until they get assembled in a software.

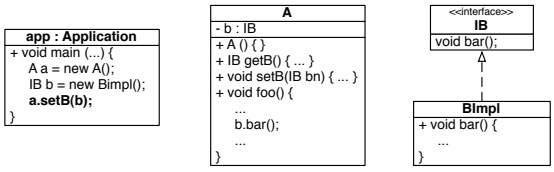

Figure 2: A decoupled implementation in OOP

In approaches such as EJB, Fractal and ArchJava, decoupling relies on the so called *dependency injection* technique [Fow04]. Figure 2 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 class A only references interface IB (type of the B instance variable and related methods). The decision to use class BIMPL and to connect its instances to instances of A is made by the developer of application APP. This solution also enables to dynamically change BIMPL by another implementation. But some other technics are not dynamically adaptable as we will see in the next section.

2.2. Adaptability

Adaptability is the ability to revise an assembly at run-time. This can be performed 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].

EJB and CCM do support only static connections. In order to change an architecture, developers have to stop the application. ArchJava does support dynamically creating and connecting components. However, 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.

2.3. Unplanned Connections

Unplanned connections refer to connections between components that were developed in different contexts, or by different component developers. These connections are decided by the application developer. So, components that have compatible functionalities should be connectable, even-though their developers didn't plan such connections. Indeed, it is impossible to a component developer to foresee all possible connections to components he develops.

While predicting all connections is not possible, COP should still 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.

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. An example of such contract, though in an OOP context, is provided in Figure 2. Definition of class A states that an instance A requires to be connected to an object compliant with interface IB. Definition of class BImpl states that every instance is compliant with interface IB. Indeed, compliance with interface IB is the contract in this example.

Most COLs are based on Java and rely on names and types as discussed above. As shown on Figure 3, 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 components were developed by different people. Therefore, in this context, application developers should share some ontology and agree about names and types.

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

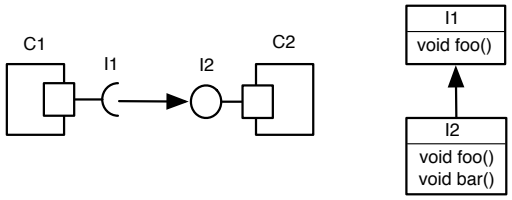

Figure 3: The problematic typing relation of independently developed components

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.

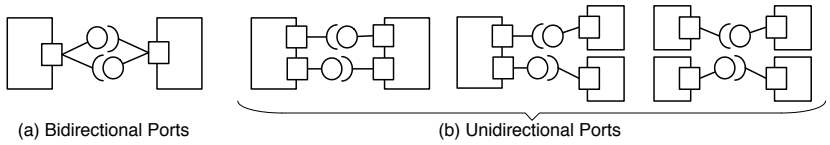

Figure 4: Unidirectional ports better support reuse by enabling unplanned connections

Another issue related to unplanned connections is the structure of ports. In the UML component model, ports can be bi-directional. A bidirectional port can only be connected to another bi-directional compatible port as shown on figure 4(a). This rule constraints the set of possible connections and goes against the unplanned connections principle. Indeed, with bidirectional ports, the unicity of the connected component is enforced. On the other hand, unidirectional ports offer more reuse opportunities since they enable multiple possibilities for connections as shown in figure 4(b).

2.4. Encapsulation and Communication Integrity

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

It is important to preserve encapsulation for two reasons. The internal structure of components may change, possibly at run-time in the case of adaptive components [GBV08, GBV06, DL03]. 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 do preserve encapsulation.

Ensuring communication integrity is a challenging issue. Consider the Fractal [BCL+06] component model and Julia, its OO-based implementation in Java. 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 supposed to be 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 5 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 5 shows the object counterparts of components C1, C2 in Julia. Through the established connection, $C1_{content}$ is allowed to make a service invocation through IC1 and IS2, in order to have it eventually executed by $C2_{content}$. Assume that service `setX` is provided by component C2 through interface IS2. When it executes its `setX` service, $C2_{content}$ may store the argument it receives which is a reference to the $C1_{content}$ object (`this` has been passed) in this example. If $C2_{content}$ does so, it may use it later in the program execution to directly communicate with the $C1_{content}$ object. We face here a violation of the communication integrity since it "short-cuts" the component interfaces.

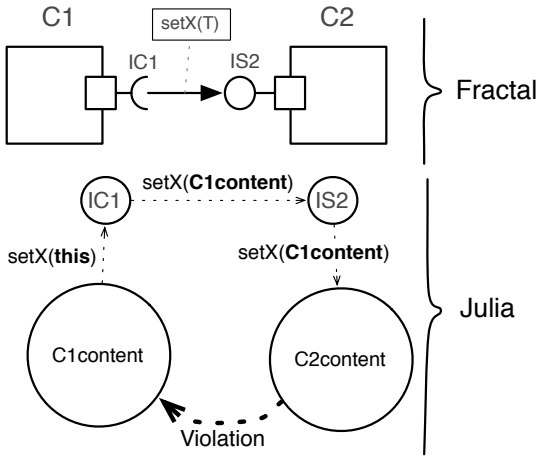

Figure 5: Violation of the communication integrity in Julia [LCL06]

This issue has been identified and solved in Julia by supervision mechanisms at runtime [LCL06]. But it brings a runtime overhead to check the validity of communications whereas it would be better to add a mechanism that prevents communication violations. 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 don't discuss this case.

specific type system named *AliasJava* [Ald03, ACN02]. This solution ensures that components can communicate only through connections that are declared by developers. Data flow is constrained and controlled based on annotations that developers put on variables to express precisely the expected behavior. Thus, 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).

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 an object can be wrapped by several layers of filters that process incoming or outgoing messages. There exist different pseudo-variables to reference the current object in its own computations. By choosing the appropriate pseudo-variable, developers make an explicit decision whether to short-cut the filters or not.

2.5. Uniformity

Most of the existing COLs and approaches to COP are based on object-oriented languages. Some, such as Fractal and EJB allow designers to reason upon components. However, 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. There is often some support to components based on code generation. While it has the nice property of reducing the amount of code typed by developers, 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 would ease code writing, code generation from ADL specifications and reverse engineering because the concepts are close enough. Only COLs offer such a continuum between design, implementation and even runtime.

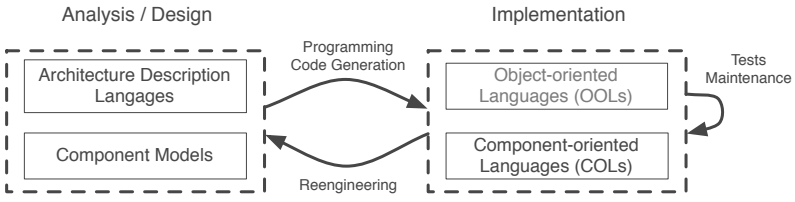

Figure 6: The need for Component-Oriented Languages to reduce the gap between design and implementation.

ArchJava is a step forward since it is a COL. However, while it does bridge the gap between design and implementation, 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);
   }
}

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

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

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

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.

In our previous work, we proposed two different solutions to this non-uniformity issue by integrating components and objects. In Scl [FDH08] we proposed a COL based on the main concepts of components and connections. Plain old objects are wrapped to be considered as regular components. In Clic [BF09], we adopted an opposite approach where we unified components and objects. Indeed Clic lives symbiotically with Smalltalk, since each Clic component is a Smalltalk object.

2.6. Summary

Regarding the open issues presented, we think that a COP approach should fulfill the five following requirements:

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 weren't 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. (R5 - Uniformity)

These requirements were partially addressed in our COL Scl. In the next section, we propose an extension to Scl that completes our previous work and addresses the five requirements listed above.

3. Revised Scl

The design of Scl has originally 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 which have evolved since [FDH08]. We introduce here optional interfaces and a new binding mechanism which eases the use of connectors and composites. These evolutions are a consequence of the better identification of the requirements presented in section 2.

3.1. 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.

Ports are dedicated to support connections and service invocations. Connection validation can be checked using interfaces. Indeed, an interface corresponds to the contract of a port. In Scl, we only focus on syntactic contracts. 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. It's worth that a port may have no interface. In this case, it accepts connections to any other port, regardless of the interface of the latter.

The use of unidirectional ports and the set inclusion relation for interface compatibility contribute to enable unplanned connections (cf. Section 2.3) in Scl. The following example shows the declaration of a TCPServer component descriptor in Scl[4]. 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.

[4] Scl uses a Smalltalk compliant syntax.

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

Scl supports *multi-port* to deal with multiple related connections. A *multi-port* is an indexed collection of ports (all required or all provided). Ports of a same kind (required or provided) can be grouped in a *multi-port* and therefore dynamically and automatically managed. Indeed, a multi-port is a collection with unlimited number of ports. New ports can be added to the collection upon need.

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.

3.2. Bindings and Connectors

As stated above, components can only be handled through their ports. Thus, 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, that is, their interfaces match as described in section 3.1.

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 `B1`, if we attempt to make `P` the source of another binding `B2`, an exception is raised. The developer should first explicitly unbind port `P` before using it as source for `B2`.

A novelty in this revisited version of Scl is that the source and the target ports can be of any kind. Thus, 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 example below illustrates a binding. It links a required port (*RequestProcessing*) to a provided one (*Processing*) of two newly instantiated components. This binding ensures that all service invocations that reach the *RequestProcessing* port will be redirected to the *Processing* port.

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

Bindings correspond to simple communication routes between two components. In order to express complex interactions involving two or more components, developers are provided 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. As shown by Figure 7, in its general form a connector has two multi-ports. The *sources* multi-port

contains provided ports through which the connector receives invocations. It then intercepts incoming invocations, adapts them and transmits them through its required ports that belong to the *targets* multi-port. A typical use of connectors is for connecting ports with mismatching interfaces. It plays the role of an adaptor and converts invocations in order to fit the target port interface.

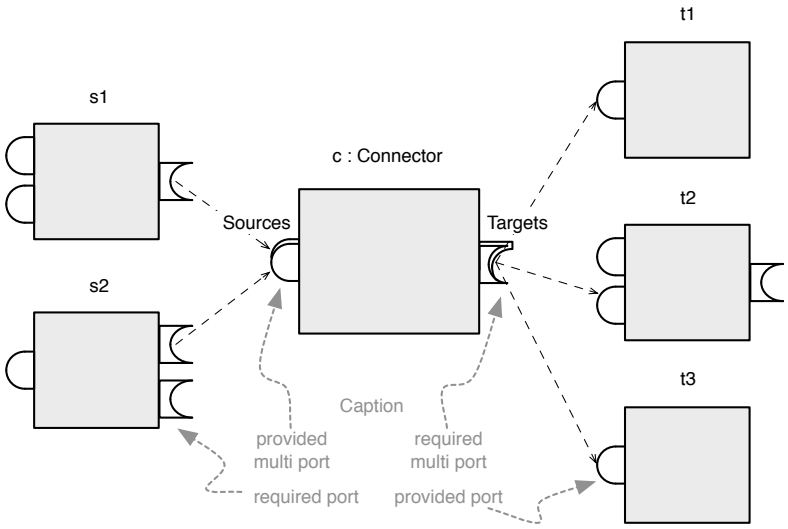

Figure 7: General Form of a Connector

3.3. 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. Otherwise, the invocation fails raising an exception.

For provided ports, the invocation handling is 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. If the provided port that receives the invocation is not the source of any binding, the pre-condition of second rule is checked. 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. If none of the above rules apply, then the third rule pre-condition is checked. The third rule is applicable if the component that owns the provided port implements a *glue service* for this port. A glue service is

analogous to the `doesNotUnderstand:` method in Smalltalk but, it is specific to a single provided port. Each time an invocation received through port does not match a service implemented on its component, the glue service is automatically executed. A component is therefore able to deal with service invocations it receives even though it does not have a directly matching implementation.

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.

3.4. 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 8 shows an example of composite instance of TCPServer. This example shows that internal ports are similar to instance variables in Smalltalk. 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.

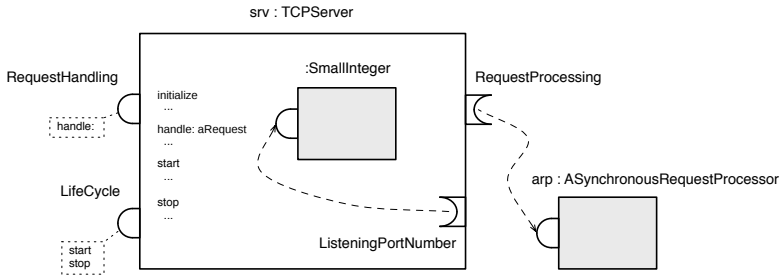

Figure 8: Example of composite with external and internal ports.

3.5. Internal Ports and Self-References

In SCL, every service invocations is achieved through a port. In our former work on SCL, the encapsulation could be violated by passing an internal port as argument or a result for an invocation.

To improve encapsulation and communication integrity (Requirement R4), we extended SCL 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[5]. This applies to the special internal port *Self*.

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

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 9 shows that the internal `kill` service can be invoked through the *Self* port.

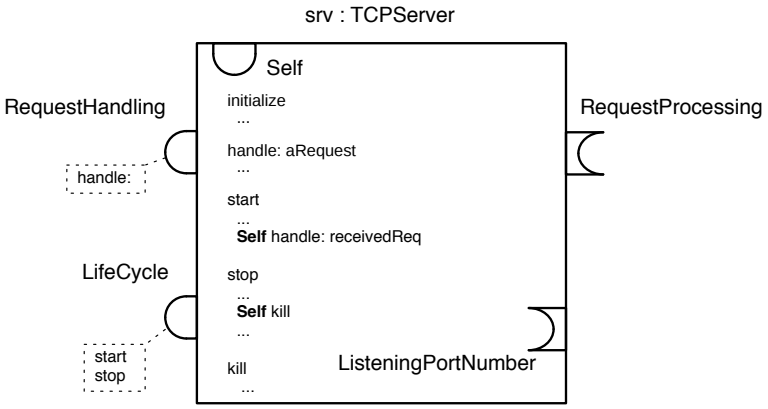

Figure 9: Some uses example of *Self* port.

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 9 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 case, if Composite2 implements a matching service, it will be executed, otherwise the invocation is delegated to the subcomponent.

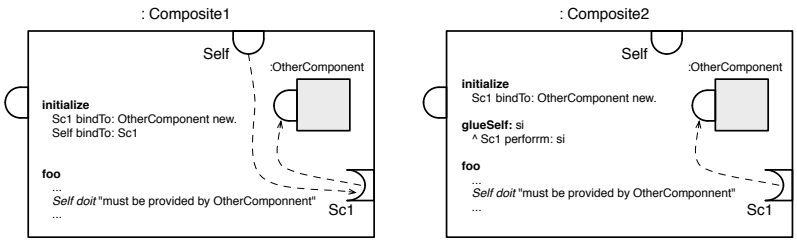

Figure 10: *Self* port can be delegated or even attached to a glue service.

These examples show that the developer is really able to control service

invocation flow with bindings. Using connectors, even more complex examples can be set up.

3.6. Argument Passing by Connection

In section 3.5, we introduced a solution to avoid exposing the internals (internal ports and subcomponents) of a component to other components. To complete our encapsulation enforcement and to ensure communication integrity, we modified the Scl service invocation to prevent connections from internal ports to external components received as parameters or invocation result. Forbidding such a connection is important, because otherwise we end up with a sub-component that is already visible to external components.

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. At the end of a service execution, all *Args* ports are unbound.

Figure 11 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 this TCPServer invokes a `process:` service through its required port *RequestProcessing* passing the aRequest as an agument. The second step shows that invocation is transmitted through the binding of *RequestProcessing*. The third step apply before executing the `process:` service implemented by the ASynchronousRequestProcessor component. All arguments passed in the service invocation are bound[6] to the *Args* ports of this component. The service is then executed in the fourth step before unbinding the *Args* ports.

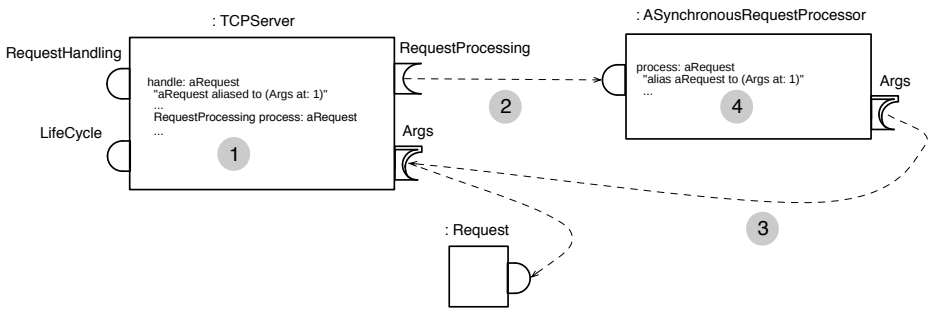

Figure 11: The four steps of arguments passing by automatic connection.

This arguments 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.

[6] Arguments are always ports in Scl.

In order to deal with situations such as recursive invocations and currency, the SCL interpreter manages a collection of argument bindings. In case of a recursion, 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. Once the latest invocation returns, the interpreter restores the bindings of *Args* of the previous service invocation. A similar solution enbales dealing with concurrency.

3.7. Summary

In this section we presented SCL a COL that was initially introduced in [FDH07]. Compared with our former work, we introduce multiple enhancements and extensions such as: bindings between ports of any kind, service invocation handling, and argument passing by connection to name a few. These revision of SCL was driven by our goal to better match requirements for COP. We list below each of the five requirements identified in Section 2 and we discuss how they are fulfilled by SCL.

R1 - Decoupling. In SCL a component description only references ports it declares. Contracts of ports are expressed as interfaces. But, matching interfaces when binding ports, only relies on comparing names and parameters of declared services.

R2 - Adaptability. Connections in SCL can be either simple bindings or complex connectors. In both cases, they can be created or destroyed at runtime, enabling thus the adaptation of assemblies.

R3 - Unplanned connections. Different facilities in SCL enable connecting components that are semantically compatible, but with different interfaces. First, SCL eases the creation of adapting connectors. Moreover, interfaces are not mandatory. 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 all ports are unbound after the invocation returns. Therefore, SCL prevents connecting a component's internal ports to external components.

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.

4. Conclusion and Perspectives

The contribution of this paper is twofold. First, we identified five requirements that should be satisfied 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. 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 consist 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 in order to take advantage of the Smalltalk tools and facilities to support COP. This study will be validated in the context of a Multi-Robots System. Our intent is to embed components on a set of robots that have to collaborate in order to accomplish some mission.

[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.

[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.

[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.

[FDH07] Luc Fabresse, Christophe Dony, and Marianne Huchard. SCL: a Simple, Uniform and Operational Language for Component-Oriented Programming in Smalltalk. In De Meuter, Wolfgang, editor, *Advances in Smaltalk, Proceedings of 14th International Smalltalk Conference (ISC), September 4–8, 2006*, volume 4406, pages 91–110. LNCS, Springer-Verlag, April 2007.

[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.

[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.

[LCL06] Marc Lger, T. Coupaye, and Thomas Ledoux. Contrle dynamique de l'intgrit des communications dans les architectures composants. In S. Vauttier R. Rousseau, C. Urtado, editor, *Langages et Modles Objets*, pages 21–36. Herms-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.

[Obj10] Object Management Group. UML 2.3 Superstructure, may 2010.

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

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

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