Available online at www.sciencedirect.com
ScienceDirect
Computer Languages, Systems & Structures 34 (2008) 130 – 149
COMPUTER LANGUAGES, SYSTEMS & STRUCTURES
www.elsevier.com/locate/cl

Foundations of a simple and unified component-oriented language

Luc Fabresse*, Christophe Dony, Marianne Huchard

Lirmm, UMR 5506 CNRS et Université Montpellier II, 161, rue Ada, 34392 Montpellier Cedex 5, France

Received 15 December 2006; accepted 1 May 2007

Abstract

Component-oriented programming (COP) is actually a key research track in software engineering. A variety of component-oriented languages (COLs) have been proposed with new or adapted abstractions and mechanisms to support this new paradigm. However, the proposed features vary quite widely from one proposal to another. There is a need for a closer analysis and synthesis of these features to really discover the new possibilities of COP. In this article we present SCL, our proposition of simple language dedicated to COP. Through the presentation of SCL, we discuss and compare the main features of COLs such as component class, component, interface, port, service or connector. But these features are not enough to build a COL. Indeed, unanticipated connection of independently developed components is one of the key issues of COP. Most approaches use language primitives or connectors and shared interfaces to connect components. But shared interfaces are in contradiction with the philosophy of independently developed components. The approach of SCL is to provide a *uniform component connection* mechanism based on special components called *connectors*. SCL also integrates *component properties* which enable connections based on component state changes with no requirements of specific code in components.

Keywords: Component-oriented language; Component connection; Connector; Component property

1. Introduction

Component-based software engineering is widely investigated by research and industry. This interest is driven by the promise of improving current software development practices, such as reusability and extensibility, in significant ways [1,2]. Although many models, languages, and tools have been proposed, it is still difficult to apply component-oriented programming (COP) in practice. Most of proposed languages, such as UML 2.0 [3] or WRIGHT [4,5], are not executable and dedicated to software specification. COP is currently carried out using object-oriented languages which do not offer specific abstractions to ease COP and have to be used in a disciplined way to guarantee a COP style.

Component-based software engineering needs component-oriented languages (COLs) as well as to transform models [6,7] into executables or to write programs by hand. A COL must offer specific abstractions or mechanisms to write component-based programs [8]. Among the approaches on components, some programming languages have

* Corresponding author. Tel.: +33 467418578.
E-mail addresses: fabresse@lirmm.fr (L. Fabresse), dony@lirmm.fr (C. Dony), huchard@lirmm.fr (M. Huchard).
URL: http://www.lirmm.fr (L. Fabresse).

1477-8424/$ - see front matter © 2007 Elsevier Ltd. All rights reserved.
doi:10.1016/j.cl.2007.05.002

been proposed such as ComponentJ [9], ArchJava [10], Julia/Fractal [11], Lagoona [8], Piccola [12], Picolo [13], Boxscript [14], Keris [15], or Koala [16], in order to support COP. These languages have brought many new or adapted abstractions and mechanisms such as connection, composition, port, interface, connector, service, module, message but their interpretation vary quite widely from one proposal to another. This is quite normal with such an emerging domain, but there is a need for a closer analysis: which mechanisms are essential (basic) and cannot be removed, which ones are (eventually) redundant? Which are the key ones to achieve component composition? To a larger extent, all these questions raise the issue of knowing which constructs and mechanisms are the main identified features of component orientation (by analogy with object orientation).

In this paper, we present SCL that stands for Simple Component Language which is the result of our study and research about COL. On one hand, SCL is built on a minimal set of concepts applied uniformly in order to ease the understanding of the key concepts of COP. The main features of SCL come from existing languages but we argue that we chose the fundamental ones. On the other hand, SCL integrates a new powerful, extensible, and uniform component connection mechanism that addresses one of the key issues of COP which is the unanticipated connection of independently developed components. In existing languages, the connection mechanism is either a fixed language primitive such as in Fractal [11] or relies on first-class entities named *connectors* [17,18] which represent connections such as in ArchJava [10] or Sofa [19]. In SCL, a connector is a kind of component dedicated to the adaptation of the communications between components. The SCL connectors offer better decoupling between the reusable business code inside components and the gluing connection code inside connectors. The SCL connection mechanism is based on connectors and enables independently developed components to communicate following different protocols without requiring any special code in components. For example, we propose connectors that offer some possibilities available in aspect-oriented programming [20]. SCL also provide connectors to establish communications based on the publish-subscribe protocol between components without requiring any special code in the publisher or the subscriber component. This is possible because SCL components are defined using the concept of *property* to externalize component state without breaking component encapsulation. Properties are the support of a new kind of component communication based on changes of property state. We choose Squeak, a Smalltalk implementation, to implement SCL because it is a dynamic language that offers a suitable meta-object protocol that can be easily extended and because we want to provide an easily extensible language.

The paper is organized as follows. Section 2 presents the general context of COP. Section 3 discusses if a class-based or a prototype-based approach is suitable for a COL. Section 4 motivates the choice of the core features of SCL. Section 5 explains the service invocation mechanism. Section 6 describes how components can be connected. Section 7 shows that separation of concerns is possible in SCL. Section 8 explains why publish/subscribe communications are an issue in COP and proposes a solution based on properties. Section 9 describes the current implementation of the SCL prototype in Squeak. Section 10 compares SCL features to those existing ones in various COLs and presents some related work. Finally, Section 11 concludes and presents future work.

2. COP: what, why and how?

COP is based on the idea stating that a software can be built by plugging pieces of software called *components*. The term "component" has different meanings to many different people depending on the perspective taken. For example, design patterns [21], functions or procedures [22], modules [8], application frameworks [2], object classes [23], and whole applications [24] can be considered as components. Similarly, there are many different definitions for the term component given in the literature [2,25,26]. The component definition reached by consensus is: "A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties" [2].

Originally, researches on component-based software development (CBSD) have been driven by the will of reducing software development costs by increasing software reuse. Indeed, instead of developing new components from scratch, CBSD recommends the reuse of existing components that have already been developed and tested. This idea is expected to reduce development costs because it reduces the development time. A second motivation for CBSD has emerged, the will of reducing the evolution cost of a software. This motivation is now more important than the original one because software are not developed one time, but continuously developed in order to correct bugs (maintenance) or to add new features (extension). Software evolution is a challenge because of new factors such as the growing software size or the software distribution. Component-based applications, built out of interconnected components, are

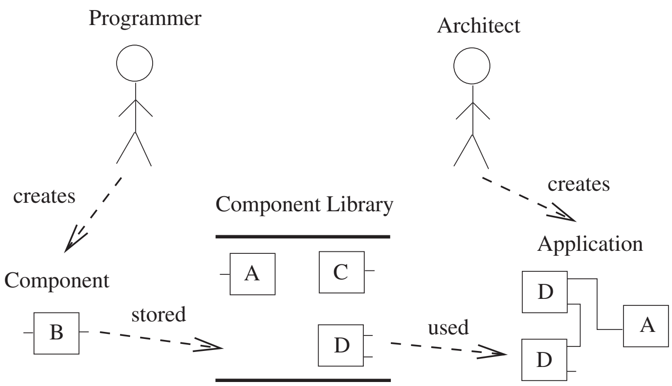

Fig. 1. A high-level view of component-based software development.

expected to be easier to evolve. This is because the software evolution of component-based software relies on decoupled components that can evolve independently. This property is named the *independent extensibility* [2] of component-based software.

New software development methods dedicated to CBSD, such as Kobra [27] or Catalysis [28], focus on the reuse in the earlier steps of the development process. Reuse at the design phase can be considered from two different complementary perspectives: *design for reuse* or *design by reuse*. Design for reuse deals with identifying, specifying reusable elements, and integrating them in a reuse system. Design by reuse aims to define new systems engineering processes and develop tools supporting systematic reuse of components to build new systems. Fig. 1 shows that these two complementary concerns are generally targeted by two different actors of the development process: the programmer and the architect. A COL must offer mechanisms for both of them, without forgetting their respective role: the programmer builds reusable components (for reuse) and the architect builds applications by reusing components (by reuse).

3. COLs: prototype-based or class-based languages?

In object-oriented languages, the terms "class" and "instance" allow programmers to refer without ambiguity, respectively, to object descriptions in code and to objects themselves as runtime entities. Although many component-based languages are generally built on a class/instance conceptual model, few of them specify the terms to denote, respectively, component descriptors (classes) and component themselves (objects). For example, the two keywords `component class` in ArchJava and `component` in ComponentJ both denote a component class which can be instantiated. In ArchJava, a component class is an extension of the Java class concept used to define a type of component. A few COLs have been proposed with a prototype-based model (i.e. without descriptors). For example, a prototype-based language has been proposed [29] on top of Java in order to provide primitives to dynamically build, extend, and compose software components from Java objects. We think that the arguments for (or against) the use of classes are similar in the component and object worlds and that both approaches are worth to be considered [30].

In SCL, we have chosen a class/instance approach but we clearly distinguish these two concepts. A *component* is a runtime entity and it is an instance of a *component descriptor*. Component descriptors are written by the *component programmer* in order to create off-the-shelf reusable pieces of software while the *software architect* creates an application by choosing and instantiating some component descriptors and then connecting components (i.e. instances).

4. Basic component structure

It is largely admitted that a "component can only be accessed through well defined interfaces" [2]. *Component interfaces* shield the component from its environment and increase its context independence needed by the design for reuse. Component-based languages propose different concepts to describe component interfaces such as services, ports, interfaces, protocols, etc. In different component languages, these words may have different meanings. For example, in Fractal [11] or Enterprise JavaBeans [31], the port and interface concept are mixed that is why they only speak about interfaces. In UML components [32], both port and interface concepts exist such as in ArchJava [10] where interfaces are called port interfaces. That is why we choose to clearly explain the choices that we made for SCL. In SCL, a component provides or requires *services* through *ports* described by *interfaces*.

4.1. Services

A component provides functionalities which are named *services*. Basically, a service is a subprogram defined in a component, such as a method in the object-oriented model. The main difference between a component and an object is that a component also expresses the services that it requires from other components in order to provide its own ones. Thus, a component has two sets of services: its *provided services* and its *required services*. For example, a password manager component may provide services such as `generatePwd:`, `generateALowerCasePwd:`, `isValid:` and require a `getRandomNumber` service.

4.2. Ports

"A component is a static abstraction with plugs" [33]. Ports represent these plugs and are the interaction points [2] of components. This means that all that is possible on a component, such as service invocation, have to be done through its ports. The port construct is present in almost all component models but with different semantics. In component models that support ports, they are either *unidirectional* or *bi-directional*. Through unidirectional ports such as in ComponentJ [9] or Fractal [11], a component provides or requires a set of services through its ports. In ArchJava [10] or UML 2.0 [32], ports are bi-directional and a component both requires and provides services through each of its ports.

Unidirectional ports allow the programmer to group some services in a set and require or provide this set. Two kind of unidirectional ports are distinguished: *required ports* through which the component requires a set of services, and *provided ports* through which the component provides a set of services. Ports help the programmer to group related services and then defines view points or security policies. Required ports define view points for the component on its environment while provided ports define view points on the component for its environment. A port also defines a security policy because a component that communicates with another component through one of its ports can only access the services accessible through this port. Grouping some required services on required port is also used by programmer to express that these services should be provided by the same component. For example, a component that requires the services: `pop` and `push:` through one of its required ports expects that these two services will be executed by the same component.

Bi-directional ports offer the same capabilities than unidirectional ones but they allow the programmer to define more complex point of views than unidirectional ports. Indeed, a bi-directional port describes the role that the component could accomplish in a collaboration. For example, a component may require a `getPwd` service and provides `open` and `close` services through the same port. This port expresses that the two services `open` and `close` will be provided to a component that itself provides the `getPwd` service. It is possible to express more accurately the dependencies between services using bi-directional ports.

We choose to integrate unidirectional ports in SCL. On the one hand, this is because they are simpler to understand and to use. The programmer defines services in the component implementation and chooses to provide some of them through the provided ports of the component. In SCL, a service may be provided by more than one port. On the other hand, this is because it is possible to construct bidirectional ports using unidirectional ones as we will see with the property concept of SCL in Section 8. In SCL, a port has a name (a component cannot have two ports with the same name). Fig. 2 shows the SCL code needed to create a component descriptor, declare ports and instantiate a component.

It is important to note that SCL uses the Smalltalk syntax but the constructs have not always the same meaning than in Smalltalk (it will be shown in Section 5). In the above example, a PASSWORDMANAGER is defined with three ports: the required `Randomizer` port through which the component requires a `getRandomNumber` service, the `Generator` and `Checker` ports that, respectively, provide services for generating passwords and services for checking passwords. The implementations of the component services are not shown in this example. COMPONENTDESCRIPTORBUILDER is the bootstrap component of SCL. This component is used to create empty component descriptors by using, for example, its `new:requiredPortNames:providedPortNames:` service. A component descriptor can be completed in order to describe its instances. Finally, the instantiation of a component descriptor is achieved using its `new` service.

```
"A PasswordManager component which generates passwords and
verifies that a password is not too simple"
ComponentDescriptorBuilder new: #PasswordManager
  requiredPortNames: 'Randomizer'
  providedPortNames: 'Generator Checker'.

(PasswordManager port: #Randomizer)
  requires: {#getRandomNumber}.
(PasswordManager port: #Generator)
  provides: {#generatePwd:. #generateALowerCasePwd:}.
(PasswordManager port: #Checker)
  provides: {#isValid:}.
...
c := PasswordManager new.
```

Fig. 2. Definition and instantiation of a component descriptor in SCL.

4.3. Interfaces

The word "interface" has many different meanings in the object world and the component world. For example, Fractal [11] distinguishes component interfaces and language interfaces (i.e. Java interfaces) but they only keep the term of interface which could be misunderstood. The interface concept is almost in all component models but sometimes mixed with the port concept such as in Fractal [11] or sometimes completely independent such as in UML [32]. Interfaces vary from informal text descriptions in natural languages to formal descriptions such as in WRIGHT [4]. We distinguish two kind of descriptions: syntactic and semantic descriptions. Syntactical descriptions are generally represented by a named type describing a set of method signatures such as Java interfaces [34]. Semantical descriptions are harder to define and are often based on formal theory, such as CSP in WRIGHT [4] or protocols in Sofa [35]. Protocols allow component programmers to define the valid sequences of service invocations through regular expressions. For example, if a port provides three services related to network communications, protocols can be used to describe that the `open:` service must be invoked first and one time, then the `send:` service, and finally the `close` service must be invoked to finish the interaction.

Interface compatibility: In ArchJava [36], ComponentJ [9], Fractal [11], a port that requires an interface I_1 can only be used by a port that provides an interface I_2 where the type defined by I_1 is a super-type of the one defined by I_2. Validation is achieved through typing rules on interfaces: "[…] types stand for semantical specification. While the conformance of an implementation to a behavioral specification cannot be easily checked by current compilers, type conformance is checkable. By simply comparing names, compilers can check that several parties refer to the same standard specification." [37]. Using these kind of interfaces implies that independently developed software components have to be defined using type-compatible interfaces in order to communicate. Depending on the type-compatible relation between interfaces, it will break the independently developed property of components. It is exactly what happens in most COLs that are Java extensions because they rely on the Java type system which is a named type system in which the sub-typing relation of interfaces is explicitly declared in their definition. Structural type systems [38] offer better decoupling since the subtype relationship is computed from the structure of interfaces. But structural type systems are less expressive than named type systems (such as Java). For example, expressing that "a component requires a Stack" is more expressive than expressing that "a component requires two services `push:` and `pop`" because it is not sure in the second case that it is exactly a stack that is needed. In the first case, there is a need for a global stack definition and the defined component will only be able to communicate with components that have been defined using the same stack definition such as "a component that provides the stack capabilities".

In SCL, an interface describes the possible interactions that can be achieved through a given port. Interfaces document the component and enable the automatic validation (static or dynamic) of the component uses. SCL interfaces can be as complex as we could imagine in order to capture the semantical aspects of components. The most basic and necessary interfaces are service signature sets and a programmer defines the interfaces of provided ports in order to choose the services that he wants to provide through a port. But the interfaces of required ports can be automatically set up on a component during its instantiation by analyzing the implementation of its provided services. Nevertheless, it is hidden

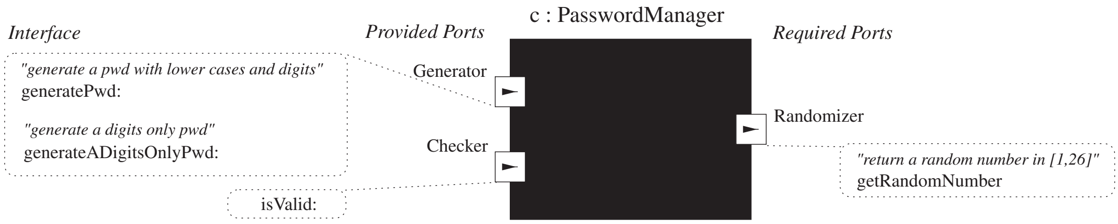

Fig. 3. The architect vision of a component c, instance of the component descriptor PasswordManager. Ports represented by squares on the component boundary. Triangles designate the direction of service invocations.

for the architect that only knows the external features of component i.e. its ports and their associated interface as shown by Fig. 3.

5. Service invocation

In Scl, components communicate by *service invocation* through their ports. The syntax and semantics of service invocation it not clearly defined in existing COLs as the message sending mechanism is in the object world. It is certainly because COLs are object-oriented language extensions such as ArchJava or ComponentJ and use the message sending mechanism.

A service invocation is similar to a message, it has a receiver, a selector, and arguments. But the receiver of a service invocation is a port that indicates through which port the service invocation is done and the selector a service names. Fig. 4 shows examples of service invocations. Since Scl uses the Smalltalk syntax, the space character is used for service invocation which is the same as message sending but the mechanism is not the same as it has been explained. If the receiver port is a required port of a component c (e.g. on line 20 of Fig. 4), the invocation will be treated by another component that will be known at connection time (cf. Section 6). The invocation of a required service of a component supposes that this component has been connected to other components that provide this service. If the receiver is a provided port of a component c, a service with a matching selector and defined on the component description of c, is executed. For example, the service invocation on line 7 of Fig. 4 will produce the execution of the provided service `isValid:`. If there is no corresponding service, the `doesNotUnderstand:` service of c is executed with the invocation as argument. As all Smalltalk objects respond to the `doesNotUnderstand:` message, all Scl components own the `doesNotUnderstand:` service.

Internal service invocations: How to invoke a service that is defined but not provided by any port of the component such as the `getRandomCharacterWithNumbers:` service? We call these kind of services *internal services*. They are useful to organize the internal code of a component but cannot be provided. Since these services are not provided nor required, it is not possible to invoke them through a port. We are not aware that some COLs have raised the question: What is `self` in a COL? In Scl, all components have an internal provided port named `self`. An *internal port* or *private port* cannot be accessed outside of its component. The *self* port of a component automatically provides all the services that are defined in its component Descriptor. The invocations on lines 5 and 14 of Fig. 4 are then regular service invocations.

Default port: We add that all components have port named `default` that provides all the provided services of the component. The services provided by this port can be restricted. This port is convenient because when a component is used as the receiver of a service invocation, this is exactly the same as if its default provided port have been used (e.g. Fig. 5).

6. Component connection

COLs propose one main mechanism for COP: the *connection* of components. A component *c* can invoke the services of another component *d*, only if *c* and *d* are connected. Component communications are possible by component connections. Unanticipated is the key-adjective attached to connection that makes component-based software worthwhile. Unanticipated means that a programmer defines a component *c* with a design for reuse goal, and he must make no other assumptions than what he declares in the interfaces of *c*. The programmer does not know the concrete components

```
PasswordManager>>generatePwd: size
"Generate a size character long password with lower cases and digits"
  | generatedPwd i |
  "..."
  i := self getRandomCharacterWithNumbers: true
  "..."
  (Checker isValid: generatedPwd)
    ifTrue: [ ^generatedPwd ]
    ifFalse: [ ^Generator generatePwd: size ]

PasswordManager>>generateALowerCasePwd: size
"Generate size character long password with lower case"
  "..."
  i := self getRandomCharacterWithNumbers: false
  "..."

PasswordManager>>getRandomCharacterWithNumbers: b
"returns a random character that is a lower case or a digit according to b"
  "..."
  i:= Randomizer getRandomNumber.
  "..."

PasswordManager>>isValid: aPwd
"returns true if the password is too easy"
  "..."
```

Fig. 4. Partial implementations of the PasswordManager services.

```
c := PasswordManager new.
c isValid: 'aaaa'.
p := c generatePwd: 4.
```

Fig. 5. Service invocations through the default port of a component.

that will be connected to c, later on by the architect. Another mechanism is proposed by COLs: the *composition* of components. Composition is used to create a new component called a *composite* out of existing components. In this section, we present these two mechanisms of connection and composition. We also explain that composition is just a subtle variation of the connection mechanism.

6.1. Connection

The connection mechanism connects components through their ports. It is often said that two components c_1 and c_2 are connected if at least one port of c_1 is connected to one port of c_2. Connecting components require to be able to connect ports. This could be achieved using language primitives or using first-class entities named *connectors*. Connectors are architectural building blocks used to model interactions among components and rules that govern those interactions [17]. The connection mechanism is provided through various forms and semantics in actual COLs. Let us examine the two solutions of ArchJava and Fractal that inspired our SCL solution.

ArchJava Aldrich et al. [36] provide a `connect` primitive that takes a set of bi-directional ports of components.

```
connect c_1.p_1, c_2.p_2, ..., c_n.p_n;
```

Semantics of this connect primitive is that when a required service of a component c_i is invoked through its port p_i, the service to be executed is searched in the set of services provided by ports $p_1, \ldots, p_n$. If the connect instruction has not

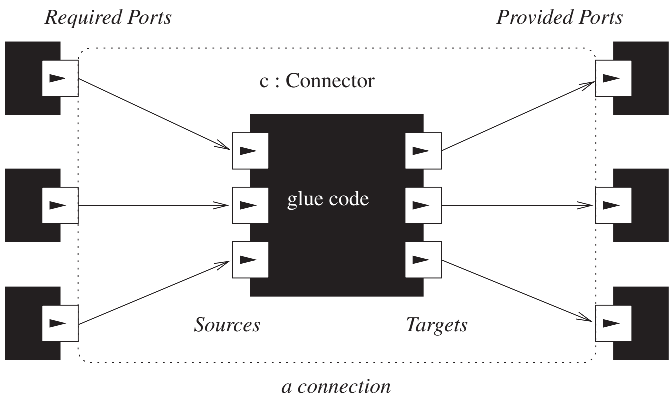

Fig. 6. The general form of a SCL connector.

failed, there is exactly one compatible service. The ArchJava mechanism is useful to easily express n-ary connections. ArchJava also allows the architect to define its own connector classes that can be used to fix the connection semantics.

```
connect c_1.p_1, c_2.p_2 with TCPConnector;
```

Fractal Bruneton et al. [11] provide a binding primitive named `bindFc` that binds one required port (named a client interface in Fractal) with one provided port (named a server interface in Fractal).

```
c1 = new C1();
c2 = new C2();
c1.bindFc("p1", c2.lookupFc("p2"));
```

More complex connections can be achieved in Fractal by using *binding components*. "A binding component is a normal Fractal component whose role is dedicated to communication between components. Binding components are also called connectors: hence Fractal does support connectors, although this concept is not a core concept here, as component or interface." [11]. In Fractal, n-ary connections require that the architect defines a binding component with exactly the right number of ports. It seems difficult with this connection mechanism to support dynamic n-ary connections.

Before presenting the SCL connection mechanism, it is important to note that a connection mechanism must address *mismatches*. Connection mismatches are an identified consequence of unanticipated connections [39]. These mismatches occur when we want to connect components that semantically fit well but their connection is not possible because they are not plug-compatible. Mismatches can be solved in whole generality by defining dedicated components as specified by the adapter design pattern [21]. Another solution is to put glue code in connections (e.g. in connectors) in order to adapt components communications. A connection mechanism must tackle this issue and makes the definition of adapters easy or useless thanks to glue code.

In SCL, the connection mechanism relies on connectors. Since we want SCL to be a simple language, we choose that a connector is a component whose role is dedicated to communication between components such as binding components in Fractal. SCL connectors help to solve mismatch problems easily and enable the definition of n-ary connections. All connectors have the same form as CONNECTOR that is shown on Fig. 6. A connector is composed of two sets of ports named `sources` and `targets`, and glue code that uses these ports to establish the connection. All the service invocations sent through source port will be treated by the glue code of the connector.

Fig. 7 shows an example of binary connection between a PASSWORDMANAGER component and a RANDOMNUMBER GENERATOR component. This connection satisfies the required service of the PASSWORDMANAGER through its `Randomizer` port, using the service `rand` provided by the RANDOMNUMBERGENERATOR through its `Generator` port. Since connectors are regular components, the architect has to define a component descriptor for the connector that is used to establish the connection. However, SCL proposes simpler syntaxes. For example, by instantiating an existing

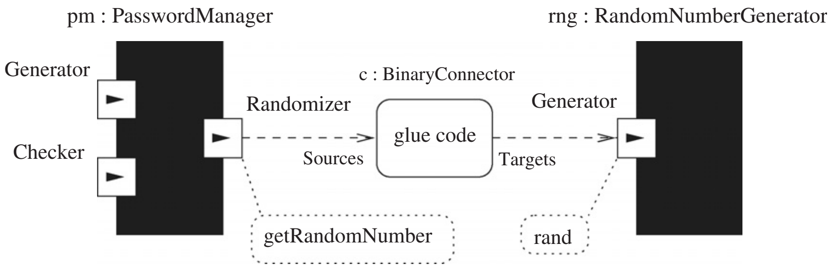

Fig. 7. A binary connection between two components in SCL.

```
pm := PasswordManager new.
rng := RandomNumberGenerator new.
c := BinaryConnector new
  source: (pm port: #Randomizer)
  target: (rng port: #Generator)
  glue: [ :source :target :serviceInvocation |
      ^(target rand * 26) asInteger
      ].
c connect.
```

Fig. 8. Connection of two components using a customized BINARYCONNECTOR.

```
c1 := C1 new.
c2 := C2 new.
(c1 port: #r1) connectTo: (c2 port: #p2).
(c1 port: #r1) connectTo: (c2 port: #p2) glue: [ :s :t :m | t foo].
(c1 port: #r1) connectTo: (c2 port: #p2) with: #TCPConnector.
```

Fig. 9. User friendly definitions of simple connections in SCL.

SCL connector descriptor, such as BINARYCONNECTOR, and adapting it by setting its sources, its targets and the glue code, as shown in Fig. 8.

Glue code is represented here by a Smalltalk block [40]. The parameters of this block are always the sources, the targets, and the current service invocation that has been received by a source port and must be transmitted through a target port. Of course, a BINARYCONNECTOR has only one source and one target. In the glue code of this example, the result of the `rand` service is adapted since the `getRandomNumber` is expected to return a number in the interval [0, 26] while the `rand` service returns a number in the interval [0, 1]. Despite the fact that this is a simple example, it is important to note that connecting independently developed software components must deal with these kinds of problems. SCL connectors tackle these adaptation issues thanks to glue code that cannot be written in components. The last line of code activates the connection by invoking the *connect* service of the connector *c* which makes him execute glue code for each invocation received through one of its source ports. Fig. 9 shows some convenient syntax for simple connections. In the two first connections, a BINARYCONNECTOR is instantiated with the default behavior that forwards all service invocations coming from the source port to the target port and to return the result back. In the third connection, a TCPCONNECTOR is used to connect these two ports.

We propose in SCL a small library of connectors based on the taxonomy of software connectors [18] that has eight kinds of connectors that have been identified. For example, the BINARYCONNECTOR is a connector of the *call connector* family that is dedicated to service invocation connections. In this family, there is for example, the BROADCASTERCONNECTOR that broadcasts each service invocation to all targets, or the FIRSTRESULTCONNECTOR that returns the first non-nil result by sending invocation successively to each target. Of course, new connectors can be defined.

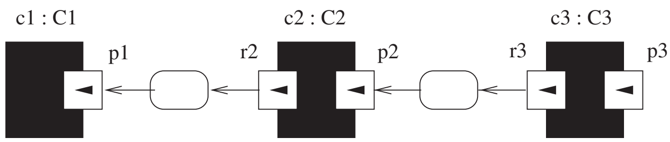

Fig. 10. Three connected components in SCL.

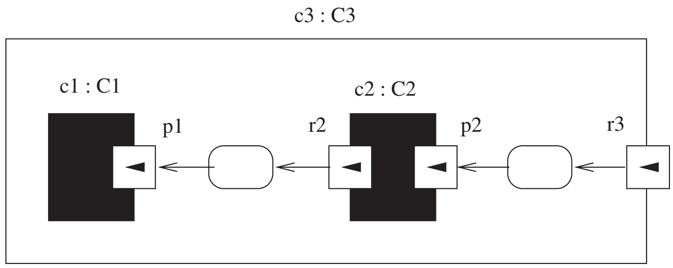

Fig. 11. A composite component.

```
ComponentDescriptorBuilder new: #C3
  requiredPortNames: 'r3'
  providedPortNames: 'p3'.

C3>>init
  |c1 c2|
  c1 := C1 new.
  c2 := C2 new.
  (c2 port: #r2) connectTo: (c1 port: #p1).
  r3 setPrivate: true.
  r3 connectTo: (c2 port: #p2).
```

Fig. 12. Definition of a composite component descriptor in SCL.

6.2. Composition

The composition mechanism is used in COLs to build composite components out of components and connections. Composite components are useful to abstract over complex systems, provide bigger reusable software entities that hide implementation details. All recent component models [9–11] provide a composition mechanism generally based on the connection mechanism to create composite. This mechanism is provided through various forms in existing languages, e.g. the `compose` primitive in ComponentJ [9], composite components in Fractal or aggregation and containment in (D)COM [24].

Composition is related to encapsulation in the component world. For example, Fig. 10 shows an example of three connected components c_1, c_2 and c_3.

If these components were objects, we could say that c_3 is a composite object that contains c_2. In the component world, this is not true because c_2 may be accessible to other components than c_3. The components used in a composite (also called its sub-components) are *internal* and *exclusive* to their composite. In our example, c_3 can be considered as a composite if c_1 and c_2 are internal and exclusively used by c_3 and if r_3 is an internal port of c_3. According to these changes, a simple drawing shift shown by Fig. 11 reveals the well known face of composite components.

Fig. 12 shows the SCL code needed to build the composite component c_3. The two sub-components c_1 and c_2 are instantiated and connected in the `init` service of c_3. The `init` service of a component is always invoked after its instantiation to initialize it. c_3 has an internal port r_3 connected to the port p_2 of c_2.

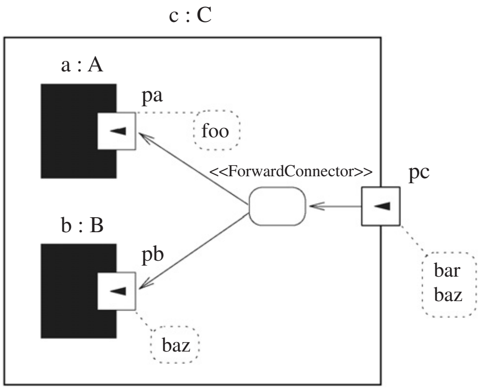

Fig. 13. Service invocation forwarding using a FORWARDCONNECTOR.

```
ComponentDescriptorBuilder new: #C
  requiredPortNames: ''
  providedPortNames: 'pc'.

C>>init
  |a b|
  a := A new.
  b := B new.
  pc provides: {#bar. #baz}.
  ForwardConnector new
    sources: {pc}
    targets: {(a port: #pa). (b port: #pb)}
    glue: [ :sources :targets :serviceInvoc |
        (serviceInvoc selector == #bar) ifTrue:[
          ^targets first perform: #foo
            withArguments: serviceInvoc arguments
        ] ifFalse: [
          (serviceInvoc selector == #baz) ifTrue: [
          ^targets second perform: serviceInvoc
    ] ] ]; connect.
```

Fig. 14. Using a connector to forward services in a composite component.

Figs. 13 and 14 show how a composite exports the services provided by its subcomponents. A FORWARDCONNECTOR is used to forward service invocations received through a provided port to another one. All mismatch problems (e.g. name conflicts, adaptation) can be addressed in this regular connector.

7. Separation of concerns in component architectures

Separation of concerns [41] principle states that a software system should be modularized in such a way that different concerns can be specified as independent as possible in order to maximize understandability and maintainability. Some concerns are difficult to encapsulate in standard software units (components or objects), such as management of transactions, logs, security, etc. To tackle the problem of the scattered code of these concerns, aspect-oriented programming [20] introduces *aspects*. An aspect is the modularization of a crosscutting concern. Two approaches are distinguished in AOP. Asymmetric approaches, such as AspectJ [42], HyperJ [43], or JAsCo [44], consider aspects as different entities from those that compose the base system (objects or components). Symmetric approaches, such as

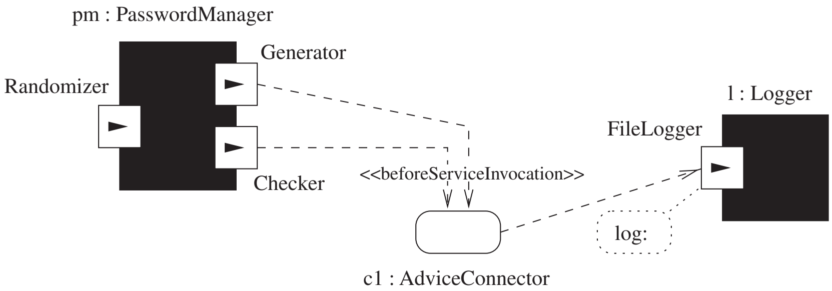

Fig. 15. A LOGGER component used as a crosscutting component using an ADVICECONNECTOR.

```
| pm rng l |
”...”
l := Logger new.

c1 := FlowConnector new
  pointcuts:
    {(pm port: #Generator) -> #beforeServiceInvocation.
    (pm port: #Checker) -> #beforeServiceInvocation}
  targets: {(l port: #Logger)}
  glue: [ :sources :targets :si |
      targets first log: ’c1 : ’, si receiver, ’ ’, si selector
  ]; connect.

c2 := FlowConnector new
  pointcuts: {(pm port: #Generator) -> #afterServiceInvocation.
      (pm port: #Checker) -> #beforeServiceInvocation}
  targets: {(l port: #Logger)}
  glue: [ :sources :targets :si |
      targets first log: ’c2 : ’, si receiver, ’ ’, si selector
  ]; connect.
```

Fig. 16. SCL code of connections that uses the ADVICECONNECTOR.

Fractal-AOP [45] or FAC [46], try to use the same entities to model the base system and aspects. This second approach is better for reusability because if aspects are modeled as components, they can be used as regular components as well as aspects. A lot of approaches try to merge aspect-oriented and component-oriented approaches in a symmetric way to benefit from the modularity properties of both approaches.

In SCL, we support some aspect-oriented features in a symmetric way. Aspects are regular components and weaving is entirely determined by connections established using special connectors and port characteristics. The *joint points*—well defined points in the execution of a program where aspects can be woven—are generally [42,44–46] method calls, method call receptions, method executions or attribute accesses. All the joint points that are available in aspect-oriented languages built on the top of object-oriented languages are not suitable for a COL. For example, in AspectJ, it is possible to specify joint points on private features of a class such as attribute accesses. Since encapsulation is key property of COLs, we only integrate in SCL the joint points that do not break this property. We choose to support the following joint points: before/after/around service invocation receptions by a port or before/after/around the connection/disconnection of a port. Fig. 15 shows an example that uses an ADVICECONNECTOR and a regular component `l` to log the service invocations received by our *PasswordManager* component.

In an ADVICECONNECTOR, each source port is coupled with a keyword (beforeServiceInvocation, beforeConnection, …). At execution time, when a service invocation arrives on a port, this invocation is transmitted to each of its connected ADVICECONNECTOR according to the standard order (around, before, after). A port is able to order its connected connectors since they declare which joint point they are interested in with a keyword. Fig. 16 shows the SCL

```
(pm port: #Generator) generatePwd: 10.
(pm port: #Checker) isValid: 'aabbcc'.

"The resulting log file contents :
c1 : #Generator #generatePwd:
c2 : #Generator #generatePwd:
c2 : #Checker #isValid:
c1 : #Checker #isValid:"
```

Fig. 17. Two service invocations and their result that illustrate the execution path.

code needed to create the connector c_1 represented on Fig. 15 and also shows the code of a connector c_2. Thanks to c_1, the `log` service of the LOGGER is executed before service invocations through `Generator` and `Checker`. Thanks to c_2, the `log` service of the LOGGER is executed after service invocations through `Generator` and before those through `Checker`. There is a conflict because multiple glue codes shall be executed before a service invocation on the same port (e.g. on lines 8 and 16). To prevent these conflicts, we introduce a priority rule stating that the glue code of the last connected connector will be executed first. This priority rule is illustrated in Fig. 17 that shows the results given by two provided service invocations of our connected PASSWORDMANAGER. These service invocations supposes that the `Randomizer` port of the PASSWORDMANAGER has been connected as previously described in Fig. 7.

8. Publish/subscribe communication between components

Triggering operations as a consequence of state changes in a component is related to observer design pattern [21] or *procedural attachments* [47]. In frame languages, it is possible to attach procedures to an attribute access which is then executed each time this attribute is accessed. These kinds of interactions are particularly used between "views" (in the MVC sense [48]) and "models". More generally, the publish/subscribe [49] communication protocol is a very useful communication pattern to decouple software entities as said in [50]: "The main invariant in this style is that announcers of events do not know which components will be affected by those events". In component-based languages, this must be done in an unanticipated way and with strict separation between the component code and the connection code to enable components reuse. However, existing proposals fail to solve these two main constraints. Connecting components based on event notifications always require that component programmers add special code in components. We identify the two following problems.

Publishers have to publish events: The component programmer has to add special code such as event signaling in components. For example, in the *JavaBean* model, the programmer has to manage explicitly the subscribers list (add and remove subscriber methods). In the CCM (corba component model) [51], the component programmer has to manage the event sending by adding a special port to his component that is called an *event source*, and sends events in the component code through this port. In ArchJava, the component programmer declares broadcast methods (required methods that return void) and invokes them in the component code to signal events. This method is then connected by the architect to multiple provided methods of subscriber components that receive the events. In all cases, the architect cannot reuse a component if its programmer has not added special code in the component to signal the event that he needs.

Emitters have to receive events: In the CCM, the component programmer has to provide its components with *event sinks* that are special ports to receive events. An event sink can be connected by the architect with one or more event sources if they share a compatible event type. This mechanism is more limiting than the ArchJava or the *JavaBeans* one where the subscriber components have only regular methods that are invoked using connections.

In SCL, there are already two ways to enable publish/subscribe connections:

(1) The component programmer integrates the event signaling in the component code. Event signaling in SCL can be done, similarly as in ArchJava, by invoking a required service in the publisher component and regular connections between publishers and subscribers. This solution supposes that the programmer has signaled specific events in the component.
(2) If no event signaling has been integrated by the programmer in the component code, ADVICECONNECTORS may be used by the architect to detect the events that he needs. For example, if the architect wants to detect when a

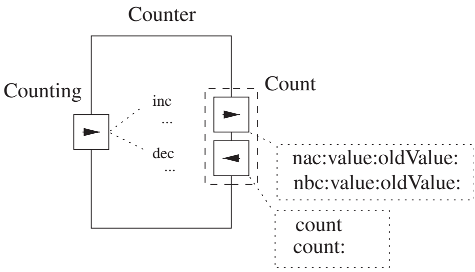

Fig. 18. A COUNTER component with a value property.

```
   ComponentDescriptorBuilder new: #Counter
2    requiredPortNames: ''
     providedPortNames: 'Counting'
4    propertyNames: 'Count'.

6  Counter>>init
     Counting provides: {#dec. #inc}.
8    self addAttribute: #value.
     Count read: [ ^value ] write: [ :nv | value := nv].
10   (self accessPortOf: #Count) setValue: 0.

12 Counter>>count
     ^(self accessPortOf: #Count) getValue
14
   Counter>>count: v
16   ^(self accessPortOf: #Count) setValue: v

18 Counter>>inc
     self count: (self count + 1)
20
   Counter>>dec
22   self count: (self count - 1)
```

Fig. 19. A Counter component class with a property.

stack becomes empty (an EmptyStackEvent), he can use an AFTERCONNECTOR on the port that provides the `pop` service and test in the glue code if the stack still contains elements to detect such situation.

If any of the above solution are available, it is not possible for the architect to establish a publish/subscribe connection without modifying the source code of the component. To prevent this issue, we integrate *properties* in SCL that export the state of a component. This property concept enhances the idea of property of the JavaBeans component model [23] with strict separation between the component code and the connection code. For example, a COUNTER component has a property named `count`. This means that it is possible to get and set a value to the `count` property of the COUNTER. An example of component with a property is depicted on Fig. 18 and the corresponding SCL code is shown on Fig. 19.

When a programmer declares a property, the component is automatically equipped with two ports: an *access port* and a *notifying port*. The property access port is a provided port that provides, at least, getter and setter services using the two blocks given during the property declaration (e.g. on line 9 in Fig. 19). The notifying port is a required port, which is used to invoke services during property accesses. These services are defined in the SCL component model. For example, the service `nac:value:oldValue:` (nac is an acronym for Notify After Change) is invoked after a property is modified with the new and the old value of the property as parameters. Another service, the `nbc:value:newValue:` (nbc is an acronym for Notify Before Change) service, is invoked before the property is modified with the current value and the next value of the property as parameters. In fact, all defined services have two main characteristics: when they are

```
gui := Label new.
counter := Counter new.

c1 := BinaryNACConnector new
  source: (counter notifyPortOf: #Count)
  target: (gui port: #Displaying)
  glue: [ :source :gui :si |
    gui displayText: (si arguments second).
  ]; connect.

c2 := (counter property: #Count) on: #notifyAfterChange
  targets: {gui port: #Displaying}
  glue: [:s :t :si | t first displayText: (si arguments second) ]
```

Fig. 20. A connection based on property notifications.

invoked (before or after the property modification) and what a connected component is able to do (nothing, prevent the modification or change the property value). Fig. 20 shows the SCL code of two connections c_1 and c_2 based on properties notifications. The first connector c_1 is created using the general syntax form and a BINARYNACCONNECTOR. This connector filters incoming service invocations on the source port and only focuses on the `nac:value:oldValue` service. After each modification of the value property of the counter, the glue code of the connection is executed and the GUI component is refreshed with the new value (the second parameter of the `nac:value:oldValue` service). c_2 achieved the same connection as c_1 with a less verbose syntax.

SCL provides different kinds of connectors such as BINARYNACCONNECTOR, BINARYNBCCONNECTOR, PROPERTYBINDERCONNECTOR ensuring that the value of the target property is always synchronized with the value of the source property. To sum up, component properties are a useful means for component programmers to directly express the external state of components instead of using syntactical conventions and for architects that can use them to connect components.

9. The SCL implementation

The actual prototype of SCL is implemented in Squeak [52]. Squeak is an open and highly portable implementation based on the original Smalltalk-80 system [40]. We choose Smalltalk because prototyping is easier and faster than in statically typed languages. It is recognized that dynamic languages offer a lot of advantages [53]. Smalltalk is also a powerful reflective language that enables deep language modifications using message interceptions, addition or modification of meta entities, etc. We also choose Smalltalk because most existing COLs are Java extensions. Using Smalltalk could reveal that existing COLs are more constrained by Java than by the component paradigm.

9.1. The SCL bootstrap

The bootstrap of SCL is done with the `ComponentDescriptor` and `Component` classes. In the first version, we would like to implement SCL without extending the class concept since we do not want to have some class features (superclass, class organization, …) in our component descriptors. We created a new bootstrapping kernel by defining `Component` and `ComponentDescriptor` as `Object` subclasses. We succeed to set `ComponentDescriptor` as the metaclass of `Component`. It was also necessary to implement our new method (`basicNew` is defined in the `Behavior` class). The problem was that all Squeak tools (code browsers, test runner, …) were not usable with our component descriptors since they were not classes. In the actual prototype, we extend classes to support additional features of component descriptors. We do not use (it is just for squeak compatibility) the features that should not be in component descriptors.

9.2. The implementation of the SCL model

Fig. 21 shows a simplified scheme reflecting the current implementation of SCL. A component descriptor is represented by a class whose subclass is `Component`. Component descriptors are instances (indirect instances

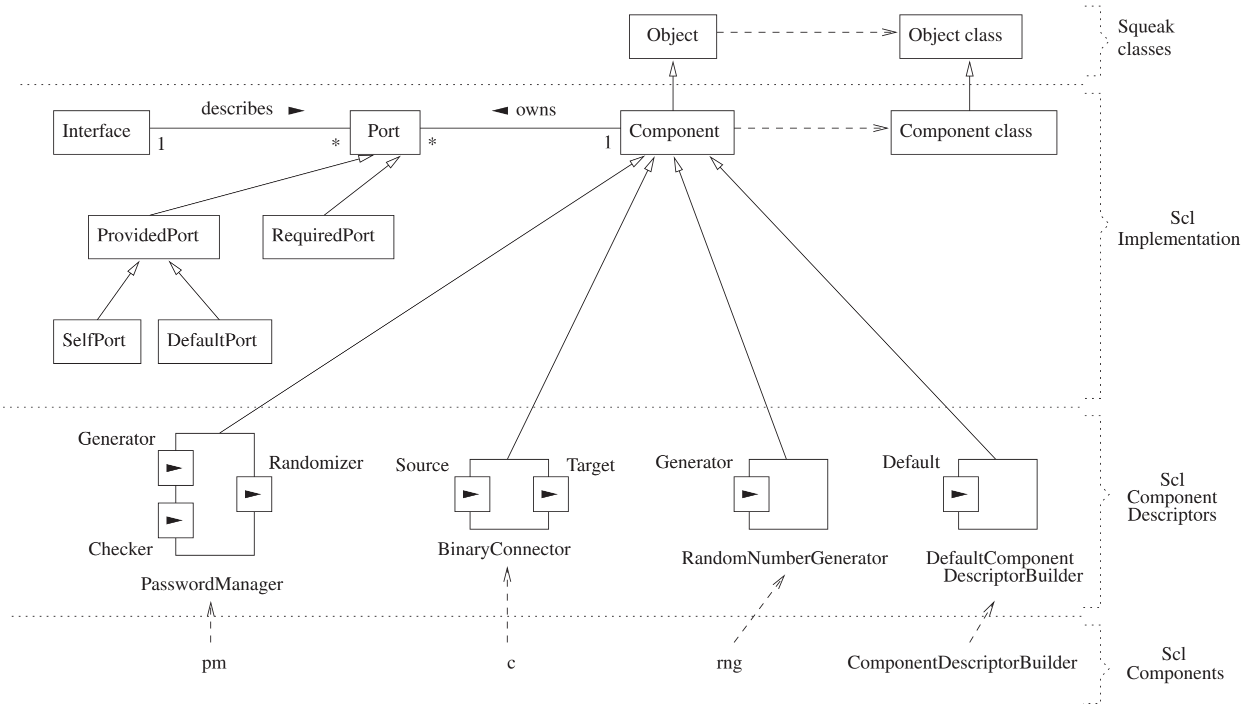

Fig. 21. Overview of the current implementation of SCL.

because of the Smalltalk parallel hierarchy) of the class named `Component class`. `ComponentDescriptor Builder` is a component, an instance of DEFAULTCOMPONENTDESCRIPTORBUILDER. All component descriptors are created using the `ComponentDescriptorBuilder` component which hides that classes are generated. Other features such as ports, interface, property are regular Smalltalk classes. It is not a problem since it is hidden to SCL programmers.

9.3. Discussion on issues in the current implementation

The first issue is related to Smalltalk that always enable the programmer to break object encapsulation by using the meta-level. If the SCL programmer uses Smalltalk methods, he is able to circumvent the SCL mechanisms. However, because we do not implement SCL with an interpreter or a compiler but directly with Smalltalk constructs, it is easier to change and evolve the implementation. This issue could be solved by creating an interpreter for SCL.

The second issue is close to the former one. Since we use the Squeak interpreter, service invocation is achieved using the message sending mechanism. This causes problems when a service invocation is sent through a port. If the service selector of the invocation corresponds to selector of one of the methods defined on the `Port` class, it is the method that will be executed and not the service of the connected component.

Another potential issue is the efficiency since we have reified all entities of SCL, and also because we do not focus this property. The execution path of a service invocation is the following: (1) the service invocation is received by the port of sending component and transmits it to the connector (it may have more than one connector), (2) the connector executes its glue code and transmits the service invocation to the port of the receiver component, (3) this port transmits the invocation to its component, (4) the component executes the method, (6) the result goes back through the same way. It is surely non-efficient but it is necessary if the glue code does some adaptation. A little optimization is possible if the default glue code (that simply transmits the invocation) is used. In this case, no connector is instantiated and the port of the sending component directly transmits the invocation to the port of the receiver component.

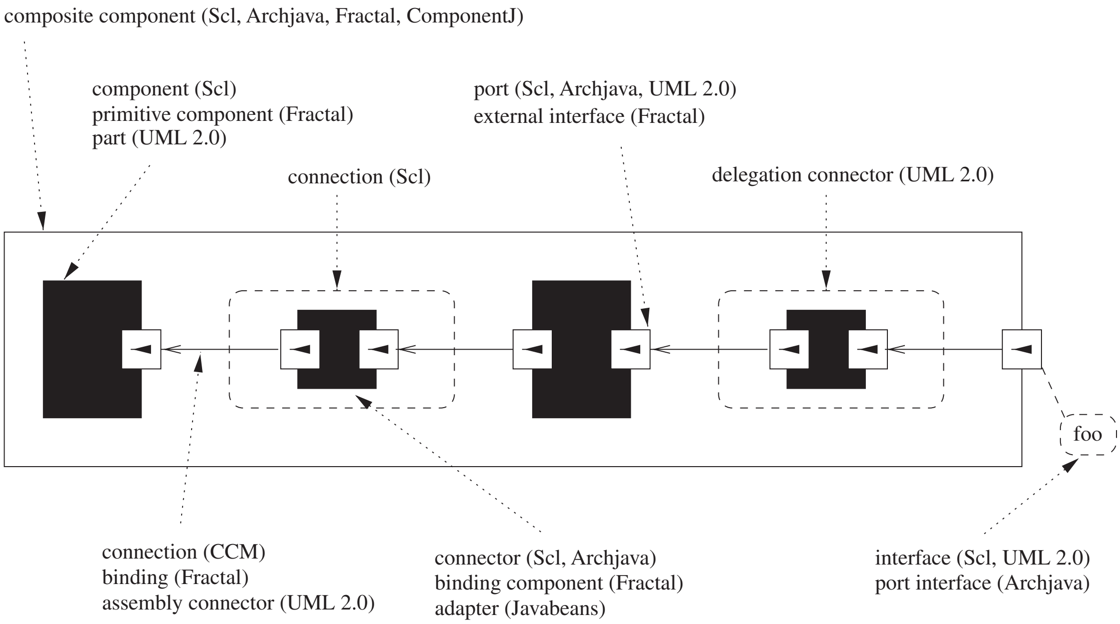

Fig. 22. Graphical comparison between SCL features and existing abstractions in some component models.

10. Synthesis and related work

Short summary of SCL *features*: An SCL component provides and requires *services*, receives, and sends *service invocations* through its *ports* which are described by *interfaces*. An *interface* is attached to a port and specifies which services are invokable through this port and the constraints that govern their invocations (service signature compliance, contracts, protocols, …). Components must be *connected* through their ports using *connectors* in order to satisfy their required services. A *connector* is a component whose role is dedicated to communication between components. There are different kinds of connectors such as advice connectors that enable using components as a crosscutting concerns. Components *properties* represent externalized state of a component and they are the support of connection based on value changes.

SCL is inspired from many existing features of COLs or models such as ArchJava [10,36], Fractal [11], CCM [51], *JavaBeans* [23], FAC [46], Fractal-AOP [45], ComponentJ [9], and others [8,13,14] SCL also integrates older ideas such as procedural attachments [47]. Fig. 22 shows a graphical synthesis (adapted from [54]) of the main SCL features compared to architectural elements available in some existing models.

The work presented in this paper is related to many different research topics.

Component-oriented programming (*COP*): Many propositions have been done to support COP using object-oriented frameworks or object-oriented language extensions. SCL tries to go a step further with new language with its own abstractions and control structures. There is few work with the same goal but Lagoona [8] can be considered as one of the first language designed with this objective. Lagoona is based on the idea that modules that contain classes and message definitions are components. Lagoona proposes a new mechanism called *generic message forwarding* that can be put in relation with service invocation in SCL. However, Lagoona lacks basic component-oriented features (such as port, connector, etc.) and is therefore more restrictive than SCL.

Understanding or teaching COP: COP is not much used compared to object-oriented programming which is actually the dominant paradigm. Although most of COLs are still research prototypes, there is a need to explain, teach, and demonstrate what is COP using simple and uniform languages as Picolo [13], BoxScript [14] or SCL. Picolo and Boxscript are frameworks, respectively, written in Python and Java.

Software architecture: Expressing software architecture in terms of connected components is the main objective of architecture description languages (ADLs) [55]. Most of actual features such as component, port, interface, connector have been originally proposed by ADLs. However, most of ADLs are not programming languages. For example, WRIGHT [4] is an ADL based on the formal language called CSP. Although a WRIGHT description is formally defined, it is not executable and must be re-written using a programming language.

Separation of concerns: Separate concerns in different modularity entity is important on one hand for reuse (a well modularized concern can be reused) and on the other hand for software architecture evolution (there is no scattered code). Separating concerns is difficult either in object-oriented languages and COLs. Aspect-oriented languages, such as AspectJ [42] or HyperJ [43] provide a suitable solution to tackle this issue by integrating some new features (aspect, advice, ...) in existing object-oriented language such as Java. COL extensions have also been proposed. Symmetrical approaches, such as FAC [46] or Fractal-AOP [45] seem to be more interesting than others because aspects are regular components. The specificity of SCL is that nothing is written in a component (no special interface has to be implemented). The architect decides to use a component as an aspect component and uses the special connector ADVICECONNECTOR. SCL does not support all AOP features because we think that some of them break the component encapsulation.

11. Conclusion and future work

COP will be only possible if new languages are proposed with concepts and mechanisms that enable the unanticipated connection of independently developed software components. In this paper, we have presented SCL a simple, uniform, and concrete COL. The SCL core results from a study of existing languages and of a selection of features that seemed fundamentals. SCL also proposes its own mechanisms such as a general connection mechanism based on connectors. Connectors are useful to solve component connection problems. In SCL, it is possible to create a wide variety of connections: standard required/provided connections, "aspect-like" connections and "publish/subcribe" connections without requiring any code in components thanks to component properties. Properties are declared by the programmer and represent external state of components. A software architect is then able to express connections based on properties notifications.

Ongoing researches on SCL are focused on three areas. First, extending the component model of SCL. For example, what is exactly a port composed of other ports (also known as composite ports or multi-ports)? The answer to this question is important because it could simplify the actual property model and connectors that have sources and target ports. Second, we plan to provide a faster and more complete version of the prototype of SCL. For example, we are wondering if it is necessary to write an interpreter in order to allow syntax changes between SCL and Smalltalk. In the new version, we want to integrate tools dedicated to COP such as a visual component editor. And finally, developing large-scale applications using SCL will certainly show us interesting results about the usability of SCL compared to object-oriented languages and the few existing COLs. This comparison would be interesting but it requires to code the same application in different languages.

References

[1] Heineman GT, Councill WT, editors. Component-based software engineering: putting the pieces together. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2001.
[2] Szyperski C. Component software: beyond object-oriented programming. 2nd ed., Reading, MA: Addison-Wesley; 2002.
[3] Object Management Group. Uml 2.0 superstructure specification. Technical Report, Object Management Group; 2004.
[4] Allen R, Garlan D. The Wright architectural specification language. Technical Report, School of Computer Science, Carnegie Mellon University, Pittsburgh; 1996.
[5] Allen R. A formal approach to software architecture, PhD thesis. School of Computer Science, Carnegie Mellon University; 1997. Issued as CMU Technical Report CMU-CS-97-144.
[6] Object Management Group. Model Driven Architecture; 2003. ⟨http://www.omg.org/mda⟩.
[7] Eichberg M. MDA and programming languages. In: Workshop on generative techniques in the context of model driven architecture (OOPSLA '02); 2002.
[8] Fröhlich PH, Gal A, Franz M. Supporting software composition at the programming-language level. Science of Computer Programming, Special Issue on New Software Composition Concept 2005;56(1–2):41–57.
[9] Seco JC, Caires L. A basic model of typed components. Lecture Notes in Computer Science 2000;1850:108–29.
[10] Aldrich J, Chambers C, Notkin D. ArchJava: Connecting software architecture to implementation. In: ICSE. ACM; 2002. p. 187–97.
[11] Bruneton E, Coupaye T, Leclercq M, Quéma V, Stefani J-B. An open component model and its support in Java. In: Crnkovic I, Stafford JA, Schmidt HW, Wallnau KC, editors. CBSE. Lecture notes in computer science, vol. 3054. Berlin: Springer; 2004. p. 7–22.
[12] Achermann F, Nierstrasz O. Applications = Components + Scripts – A Tour of Piccola. In: Akşit M, editor. Software architectures and component technology. Dordrecht: Kluwer Academic Publishers; 2001. p. 261–92.
[13] Marvie R. Picolo: a simple python framework for introducing component principles. In: Euro Python conference 2005. Göteborg, Sweden; June 2005.

[14] Liu Y, Cunningham HC. Boxscript: a component-oriented language for teaching. 43rd ACM-Southeast conference, vol. 1, March 2005. p. 349–54.
[15] Zenger M. Keris: evolving software with extensible modules: research articles. Journal of Software Maintenance and Evolution 2005;17(5): 333–62.
[16] van Ommering RC. Koala, a component model for consumer electronics product software. In: van der Linden F, editor. ESPRIT ARES workshop. Lecture notes in computer science, vol. 1429. Berlin: Springer; 1998. p. 76–86.
[17] Shaw M. Procedure calls are the assembly language of software interconnection: connectors deserve first-class status. In: ICSE '93: selected papers from the workshop on studies of software design. London, UK: Springer; 1996. p. 17–32.
[18] Mehta NR, Medvidovic N, Phadke S. Towards a taxonomy of software connectors. In: ICSE '00: Proceedings of the 22nd international conference on software engineering. New York, USA: ACM; 2000. p. 178–87.
[19] Balek D, Plasil F. Software connectors and their role in component deployment. In: Proceedings of DAIS'01, Krakow, Poland, September 2001. Dordrecht: Kluwer Academic Publishers; 2001.
[20] Kiczales G, Lamping J, Mendhekar A, Maeda C, Lopes C, Loingtier J-M. et al. Aspect-oriented programming. In: Akşit M, Matsuoka S, editors. 11th European conference on object-oriented programming. Lecture notes in computer science, vol. 1241. Berlin: Springer; 1997. p. 220–42.
[21] Gamma E, Helm R, Johnson R, Vlissides J. Design patterns: elements of reusable object-oriented software. Reading, MA: Addison-Wesley; 1995.
[22] McIlroy MD. Mass produced software components. In: Naur P, Randell B, editors. Proceedings, NATO conference on software engineering. Garmisch, Germany; October 1968.
[23] Hamilton G. JavaBeans, API Specification, Sun Microsystems, July 1997. Version 1.01.
[24] Microsoft. DCOM technical overview. Microsoft Windows NT Server white paper, Microsoft Corporation; 1996.
[25] Broy M, Deimel A, Henn J, Koskimies K, Plasil F, Pomberger G. et al. What characterizes a (software) component?. Software—Concepts and Tools 1998;19(1):49–56.
[26] Gröne B, Knöpfel A, Tabeling P. Component vs. component: why we need more than one definition. In: ECBS. IEEE Computer Society; 2005. p. 550–2.
[27] Atkinson C, Paech B, Reinhold J, Sander T. Developing and applying component-based model-driven architectures in kobra. edoc, 00:0212; 2001.
[28] D'Souza DF, Wills AC. Objects, components, and frameworks with UML: the catalysis approach. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc.; 1999.
[29] Zenger M. Type-safe prototype-based component evolution. In: Proceedings of the European conference on object-oriented programming. Spain, Malaga; June 2002.
[30] Dony C, Malenfant J, Cointe P. Prototype-based languages: from a new taxonomy to constructive proposals and their validation, In: OOPSLA; 1992. p. 201–17.
[31] Monson-Haefel R. Enterprise JavaBeans. Sebastopol, CA, USA: O'Reilly & Associates Inc.; 1999.
[32] Cheesman J, Daniels J. UML components: a simple process for specifying component-based software. Boston, MA, USA: Addison-Wesley; 2000.
[33] Nierstrasz O, Dami L. Component-oriented software technology. In: Nierstrasz O, Tsichritzis D, editors. Object-oriented software composition. Englewood Cliffs, NJ: Prentice-Hall; 1995. p. 3–28.
[34] Gosling J. The Java language specification. Boston: Addison-Wesley; 2000.
[35] Plasil F, Visnovsky S. Behavior protocols for software components. IEEE Transactions on Software Engineering 2002;28(11):1056–76.
[36] Aldrich J, Sazawal V, Chambers C, Notkin D. Language support for connector abstractions. In: Cardelli L, editor. ECOOP. Lecture notes in computer science, vol. 2743. Berlin: Springer; 2003. p. 74–102.
[37] Büchi M, Weck W. Compound types for Java. In: OOPSLA'98: Proceedings of the 13th ACM SIGPLAN conference on object-oriented programming, systems, languages, and applications. New York, USA: ACM; 1998. p. 362–73.
[38] Cardelli L. Type systems. The handbook of computer science and engineering. Boca Raton, FL: CRC Press; 1997. p. 2208–36 [chapter 103].
[39] Sametinger J. Software engineering with reusable components. New York, NY, USA: Springer; 1997.
[40] Goldberg A, Robson D. Smalltalk-80: the language. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc.; 1989.
[41] Parnas DL. On the criteria to be used in decomposing systems into modules. Communications of the ACM 1972;15(12):1053–8.
[42] Kiczales G, Hilsdale E, Hugunin J, Kersten M, Palm J, Griswold WG. An overview of AspectJ. In: Knudsen JL, editor. ECOOP. Lecture notes in computer science, vol. 2072. Berlin: Springer; 2001. p. 327–53.
[43] IBM. Hyper/J. ⟨http://www.research.ibm.com/hyperspaces⟩.
[44] Suvée D, Vanderperren W, Jonckers V. Jasco: an aspect-oriented approach tailored for component based software development. In: AOSD '03: Proceedings of the second international conference on aspect-oriented software development. New York, USA: ACM; 2003. p. 21–9.
[45] Fakih H, Bouraqadi N, Duchien L. Aspects and software components: a case study of the fractal component model. In: Huang M, Mei H, Zhao J, editors. International workshop on aspect-oriented software development (WAOSD 2004). September 2004.
[46] Pessemier N, Seinturier L, Duchien L, Coupaye T. A model for developing component-based and aspect-oriented systems. Proceedings of the fifth international symposium on software composition (SC'06). Lecture notes in computer science, vol. 4089. Berlin: Springer; 2006.
[47] Minsky M. A framework for representing knowledge. In: Winston P, editor. The psychology of computer vision. New York: McGraw-Hill; 1975. p. 211–81.
[48] Krasner GE, Pope ST. A cookbook for using the model-view-controller user interface paradigm in Smalltalk-80. Journal of Object-Oriented Programming 1988;1:26–49.
[49] Eugster PTh, Felber PA, Guerraoui R, Kermarrec A-M. The many faces of publish/subscribe. ACM Computing Surveys 2003;35(2):114–31.
[50] Garlan D, Shaw M. An introduction to software architecture. In: Ambriola V, Tortora G, editors. Advances in software engineering and knowledge engineering. Singapore: World Scientific; 1993. p. 1–39.

[51] Object Management Group. Manual of Corba Component Model V3.0; 2002. ⟨http://www.omg.org/technology/documents/formal/components.htm⟩.
[52] Ingalls D, Kaehler T, Maloney J, Wallace S, Kay A. Back to the future: the story of Squeak, a practical Smaltalk written in itself. In: OOPSLA '97: Proceedings of the 12th ACM SIGPLAN conference on object-oriented programming, systems, languages, and applications. New York, USA: ACM; 1997. p. 318–26.
[53] Nierstrasz O, Bergel A, Denker M, Ducasse S, Gälli M, Wuyts R. On the revival of dynamic languages. In: Gschwind T, Aßmann U, editors. Proceedings of software composition 2005. Lecture notes in computer science, vol. 3628, 2005. p. 1–13 [invited paper].
[54] Tibermacine C. Contractualisation de l'évolution architecturale de logiciels à base de composants: une approche pour la préservation de la qualité. PhD thesis, Université de Bretagne Sud; October 2006.
[55] Medvidovic N, Taylor RN. A classification and comparison framework for software architecture description languages. Software Engineering 2000;26(1):70–93.