SCL: a Simple, Uniform and Operational Language for Component-Oriented Programming in Smalltalk

Luc Fabresse, Christophe Dony, Marianne Huchard

SCL: A Simple, Uniform and Operational Language for Component-Oriented Programming in Smalltalk

Luc Fabresse, Christophe Dony, and Marianne Huchard

Lirmm, UMR 5506 CNRS et Université Montpellier II
161, rue Ada
34392 Montpellier Cedex 5
{fabresse,dony,huchard}@lirmm.fr
http://www.lirmm.fr

Abstract. Unanticipated connection of independently developed components is one of the key issues in component-oriented programming. While a variety of component-oriented languages have been proposed, none of them has achieved a breakthrough yet.

In this paper, we present SCL a simple language dedicated to component-oriented programming. SCL integrates well-known features such as component class, component, interface, port or service. All these well-known features are presented, discussed and compared to existing approaches because they vary quite widely from one language to another. But, these features are not enough to build a component language. Indeed, most approaches use language primitives and shared interfaces to connect components. But shared interfaces are in contradiction with the philosophy of independently developed components. To this issue, SCL provides new features such as a *uniform component composition model* based on *connectors*. Connectors represent interactions between independently developed components. SCL also integrates *component properties* which enable connections based on component state changes with no requirements of specific code in components.

Keywords: component-oriented programming, unanticipated composition, 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 in significant ways such as reusability and extensibility [23,45]. Although many models, languages and tools have been proposed, it is still difficult to apply component-oriented programming (COP) in practice. Most of these languages are not executable and dedicated to software specification such

W. De Meuter (Ed.): ESUG 2006, LNCS 4406, pp. 91–110, 2007.

as UML 2.0 [21] or architecture description such as WRIGHT [5,4]. COP is currently carried out using object-oriented languages. These languages 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 (COL) as well as transformation of models [37,12] into executables or writing programs by hand [16]. Among the approaches on components, component-oriented languages have been proposed in order to support COP such as ComponentJ [42], ArchJava [2], Julia/Fractal [8], Lagoona [16], Piccola [1], Picolo [30], Boxscript [29], Keris [48] or Koala [46]. The contributions of these languages are new or adapted abstractions and mechanisms that vary quite widely from one proposal to another such as connection, composition, port, interface, connector, service, module, message, etc. 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 propose SCL that stands for Simple Component Language which is the result of our study and research of component-oriented programming. SCL is built on a minimal set of concepts applied uniformly in order to ease the understanding of key concepts of component-oriented programming. Picolo [30] and BoxScript [29] are two languages that also target this goal of minimality for simplicity. However, SCL integrates a more powerful and extensible component composition mechanism which is one of the key mechanisms of COP. In SCL, component composition relies on first-class entities representing connections, named *connectors* [43,33]. Connectors offer better decoupling between the business code inside components and the connection code inside connectors, and thus increase the reuse of components. Some COL already propose connectors such as ArchJava [2] or Sofa [6], but SCL connectors offer more expressiveness by integrating ideas that come from aspect-oriented programming [26]. SCL also proposes the concept of *property* to externalize component state without breaking component encapsulation. Properties are the support of a new kind of component communication that is based on changes of property state. Properties ease the use of the publish-subscribe communication pattern without requiring any special code in the publisher or the subscriber. 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. Although it is also possible to implement SCL in another language, we choose to experiment COP in a dynamic context and we want to provide an easily extensible language.

The paper is organized as follows. Section 2 presents basic ideas of component-oriented programming. Section 3 details main characteristics of the SCL language: component classes, components, ports, interfaces, connectors and properties.

Section 4 presents the current implementation of Scl in Squeak. Section 5 discusses related work. Finally, Section 6 concludes and presents future work.

2 Component-Oriented Programming: What, Why and How ?

Component-oriented programming (COP) does for decoupling software entities what object-oriented programming has done for object encapsulation and inheritance, or aspect-oriented programming has done for crosscutting concerns. It provides language mechanisms that explicitly capture software architecture structures. COP is based on the idea stating that software can be built by plugging pieces of software called *components*. The term "component" means many different things to many different people depending upon the perspective taken on the development. For example, design patterns [17], functions or procedures [31], modules [16], application frameworks [45], object-oriented classes [22], and whole applications [34] are considered as components. Similarly, there are many different definitions for the term component given in the literature [7,20,45]. In this paper, we use the following definition : *"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"* [45].

Component-based software development focuses on better reuse and easier evolution. A component must be independent of one particular context in order to be reusable. Furthermore, reusing a component is better than creating it from scratch because it has already been developed and tested. The evolution and maintenance of a component software architecture may be easier than a class hierarchy. This is because of the *independent extensibility* [45] property of component-based software. Indeed, component-based applications are built out of interconnected components and each component can evolve independently.

3 The Scl Language

In this section we describe Scl (Simple Component Language). We present and motivate its main features and discuss the problems that arise when designing a COL.

3.1 Component Classes and Component Instances

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 component-based languages are generally built on a class/instance conceptual model, few of them specify the terms to denote respectively component classes and component objects. Moreover, there is no widely accepted terms in component-oriented approaches because there is not a unique definition of the component term. For example,

the two keywords `component class` in ArchJava and `component` in ComponentJ denote a component class which can be instantiated. Component classes are at the same time component descriptors, component instantiators and component method holders such as in ArchJava. Few COLs have been proposed with a prototype-based model i.e without descriptors such as in [47] where a prototype-based language has been proposed on the 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 the use (or not) of classes is similar in the component and object worlds and that both approaches are worth to be considered. In Scl, we have chosen a class/instance approach. A *component* is a runtime entity and it is an instance of a *component class*. Component classes 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 some component classes and then connecting instances i.e components. Figure 1 shows the code to create a component class and the code to create a component using the `new` message.

```
SCLComponentClassBuilder create: #MyComponent.
...
c := MyComponent new.
...
```

Fig. 1. A component descriptor and a component instance

3.2 Component Provisions and Requirements

Component interfaces and services. As stated by Szyperski [45], a component can only be accessed through well-defined *interfaces*. Component interfaces enforce explicit context-dependencies and a high-level of encapsulation. A component interface describes only a *service* or a group of services provided by a component to other components. A component provides a lot of services through different interfaces but its clients can only use those ones defined in the interface which they are connected to. Component interfaces also specify the services that are required by a component to be able to provide its own services. Basically, a service is a subprogram defined in a component, such as a method in the object-oriented model. The term service is used to refer to a high-level functionality. For example, a *Network* service is at least composed of four methods: `open:` to initialize a network connection, `close` to finish the connection, `send:` and `receive` to respectively send and receive data through an open connection. Component-based languages propose different concepts to describe component interfaces such as ports, interfaces, protocols, etc. In Scl, we choose to represent component interfaces by *ports* described by *interfaces*. We argue that these two concepts are enough to describe component interfaces.

Ports. Ports represent interaction points of a component such as in ArchJava [2] or ComponentJ [42]. The port construct has not the same definition and

characteristics in all COLs. For example in Picolo, ComponentJ or Fractal (ports are called external interfaces in Fractal), ports are *unidirectional* because they provide or require a set of services. In ArchJava or UML 2.0 [11], ports are *bi-directional* and the component invokes external services and receives service invocations through the same port. Required services through a port have to be provided by the same component. For example, a component that requires a *Network* service through one of its ports, expects services `open:` , `send:`, `receive` and `close`, will be executed by the same component. However, provided services are accessible to one or many other components. Providing and requiring services through one port may result in limiting the use of the provided services to only one component at a time. SCL integrates two kinds of unidirectional ports: those ones for accessing required services and those ones for giving access to provided services. A port has a name. A component can not have two ports with the same name. A port name is used in the code to specify through which port a service is invoked. A service is always invoked through a port by message sending (the same term as in object world is used). Syntactically, the port is the receiver but in fact, the real receiver of a message is always a component that will be known at connection time. Note that it is worth to invoke a service that the component itself defines. All components have a special internal provided port named `self` that can not be accessed outside of the component. In this context, the invocation `self foo` is equivalent to a service invocation that requires no connection to be achieved and that executes the `foo` service of current component. To sum up, an SCL component offers or requires *services*, receives or sends *service invocation*, and can be *connected* through its ports as we will see later in Section 3.3.

Interfaces. An interface describes the valid interactions through a port in order to document the component or to enable the automatic validation (static or dynamic) of the component uses and the connections. In COLs, these descriptions vary from simple ones such as informal texts in natural languages to complex ones such as formal descriptions. These descriptions are classified in two categories: syntactic and semantic.

Syntactical descriptions are generally represented using *interfaces* (such as in Java). An interface defines a named type describing a set of method signatures. Validation of the use of a port relies on typing rules. For example, a port that requires an interface I_1 can be connected with a port that provides an interface I_2 where the type defined by I_1 is a supertype of the one defined by I_2. Using interfaces implies that independently developed software components have to refer to a common standard defined by interfaces in order to inter-operate. Other solutions exist, such as structural type systems [10], that offer better decoupling between component classes. But structural type systems are less expressive than named type systems (such as with interfaces) as said in [9], *"[...] 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."*. For example, writing

that "a component requires a stack" is more expressive than writing that "a component requires two services `pop` and `push:`" but in the first case there is a need for a global stack definition.

Semantical descriptions are harder to define and are often based on formal theory, such as CSP in WRIGHT [5] or protocols in Sofa [40]. For example, protocols allow component programmers to define the valid sequences of service invocations through regular expressions. In our last example of the Network service, it is important to describe that firstly the `open:` service has to be invoked, then the `send:` and `receive` services can be used, and finally the `close` service must be invoked to finish the interaction.

Ports and Interfaces in Scl. We choose to decouple component classes and avoid global definitions such as named interfaces. This is the reason why interfaces are service signature sets in SCL and not named interfaces. But, it is possible to extend SCL to support more sophisticated interfaces as protocols. Figure 2 shows an example of component class with ports and Figure 3 shows the SCL code needed to declare it. PASSWORDMANAGER is a component class created by the bootstrap method SCLCOMPONENTBUILDER>>create: that creates an empty component class. In its internal service named `init`, the PASSWORDMANAGER is composed of three ports: `Randomizing` is a required port since it is used to invoke external services of the component; `Generating` and `Checking` are required ports because they offer some services of the component and receive service invocations from third parties. Since we do not focus on static or dynamic validation of connections, it is not mandatory to specify interfaces of required ports. The same situation happens in dynamically-typed languages when method parameters are not described by a static type. However, interfaces of provided ports are needed because they specify which services of the components are provided through the port.

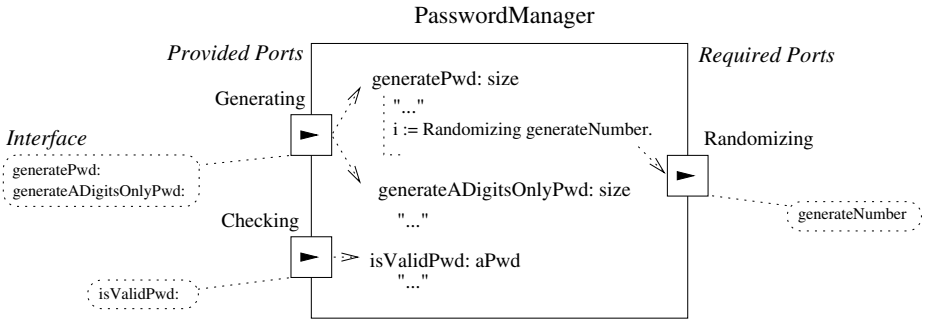

Fig. 2. A SCL component. Ports are represented by squares on the component boundary and triangles designate the direction of service invocations. Ports are described by interfaces which are service signature sets.

```
SCLComponentClassBuilder create: #PasswordManager.

PasswordManager>>init
  self addPort: (SclPort newNamed: #Randomizing requires:
      (SclInterface new with: {#generateNumber})).
  self addPort: (SclPort newNamed: #Generating provides:
      (SclInterface new with: {
        #generatePwd:.
        #generateADigitsOnlyPwd:
      })).
  self addPort: (SclPort newNamed: #Checking provides:
      (SclInterface new with: {#isValidPassword:})).

PasswordManager>>generatePwd: size
  "..."
  i := Randomizing generateNumber.
  "..."
```

Fig. 3. A component class declaration

3.3 Component Composition

There are two main mechanisms for unanticipated composition of components: *connection* and *composition*[1]. **Unanticipated** is the key-adjective attached to composition or connection that makes component-based software worthwhile. To be composable, a component definition should only state what it provides and what it needs and should make no assumption about which other concrete components it will be composed with later on.

Connection. As said in [36], "*a component is a static abstraction with plugs*". In SCL, plugs of components are their ports. The **connection** is the mechanism that connects component ports. The connection mechanism is provided through various forms in actual COLs, e.g. the `connect` primitive and connectors in ArchJava [3], the `plug` primitive in ComponentJ [42], connectors in Picolo [30] or bindings in Fractal [8]. Connections are the support for the communication between components and they enforce the decoupling between components which can not communicate if they have not been connected.

Connection mismatches are identified consequences of unanticipated connections [41]. 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 [17]. There is a need for glue code in connections to adapt components. A connection mechanism must be flexible to make the definition of adapters useless.

Connecting components could be achieved using language primitives such as plug in ComponentJ [42]. Other component models propose connections as first-class entities named *connectors* such as Sofa [6] or ArchJava [3] and most of Architecture Description Languages [32], such as WRIGHT [5,4]. Connectors are

[1] The term *composition* is used here for a mechanism that creates a new component out of existing ones.

architectural building blocks used to model interactions among components and rules that govern those interactions [43]. Unlike components, connectors may not correspond to compilation units or deployment units. In Scl, SclConnector is the most general form of connectors which is composed of two sets of ports named *sources* and *targets* and *glue code* that only uses these ports to establish the connection. SclCallConnector is the general connector dedicated to service invocation connections as shown in Figure 4.

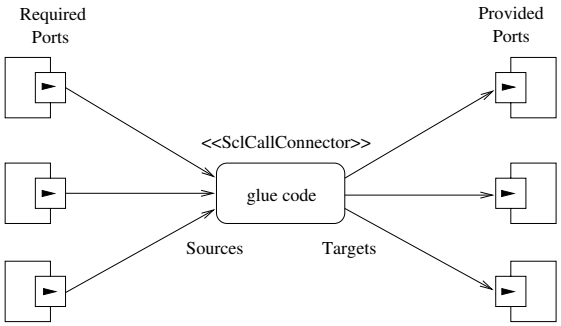

Fig. 4. The general form a Scl connector

In a SclCallConnector, sources are required ports and targets are provided ports. It is possible to define specialized connectors that provide a general purpose glue code or restrict sources and targets. Among the various existing connector types, there is one, SclBinaryConnector that restricts itself to one source and one target. Figure 5 shows an example of binary connection between a PasswordManager component and a RandomNumberGenerator component. This connection satisfies the required service of the PasswordManager through its `Randomizing` port, using the service `generateNumber` provided by the RandomNumberGenerator through its `Generating` port. Figure 6 shows the code to establish this connection.

The glue code of a SclCallConnector is a Smalltalk block whose parameters are the set of sources, the set of targets and the current service invocation (which includes the source port, the selector and parameters) that has to be performed. In the glue code of this example, the result of the `rand` service is adapted since the `generatedpassword` is expected to return a number in the interval $[0, 26]$ while the `rand` service returns a number in the interval $[0, 1]$. Despite of 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. The glue code in connectors is a good place to tackle these adaptation problems. If no glue code is specified in a SclBinaryConnector, the default behavior is to forward all services that come from the source port to the target port and to return the result. This is the same as the Fractal *bind* primitive or the ComponentJ `plug` primitive.

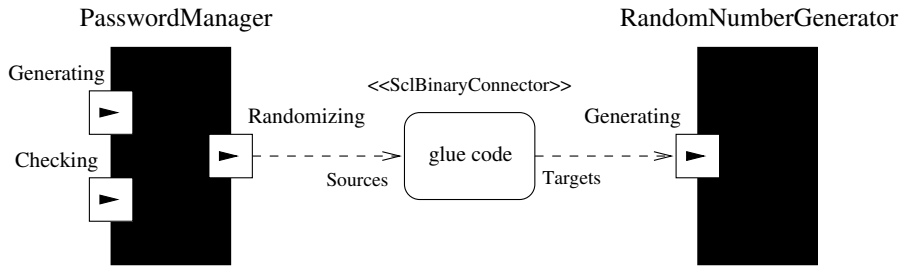

Fig. 5. A Scl connection of two components

Like the SclBinaryConnector, it is possible to build reusable connectors, such as BroadcasterConnector, that broadcasts each service invocation to all targets, or FirstResultConnector that returns the first non-nil result by sending invocation successively to each target.

```
spm := PasswordManager new.
srng := RandomNumberGenerator new.
SclBinaryConnector new
  source: (spm port: #Randomizing)
  target: (srng port: #Generating)
  glue: [ :source :target :message |
      ^(target rand * 26) asInteger
      ];
  connect.
```

Fig. 6. Connecting two components

Composition. Composition is the mechanism that builds a composite component out of components and connections. Encapsulated components are generally called sub-components of the composite. Composite components are useful to abstract over complex systems and provide a new reusable software entity that hide implementation details. This mechanism is provided through various forms in existing languages, e.g the `compose` primitive in ComponentJ [42], composite components in Fractal or aggregation and containment in (D)COM [34].

Figure 7 and Figure 8 show the architecture and the code of a simple composite in Scl. A composite component `c` instance of the component class C encapsulates two components `a` and `b` and one connection. Each instance of C *forwards* the provided port `pb` of its subcomponent `b` for external uses. This example is quite simple and more complex ones require the use of SclForwardConnector s. These kinds of connectors are used to forward externalize services of sub-components in a composite component. The sources and targets are all required or provided ports and the glue code can be used to solve problems such as name conflicts, etc. Figure 9 and Figure 10 shows this situation with a composite component that provides two services on a same port but provided by two different sub-components.

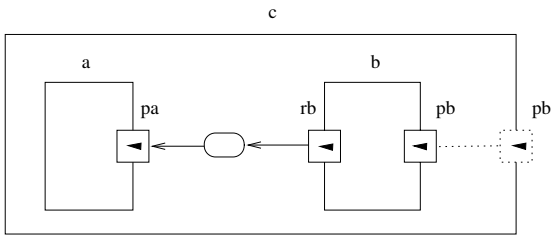

Fig. 7. A composite component that forwards a port

3.4 Separation of Concerns in Component Applications

Separation of concerns [38] 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 maintenability. 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 [26] introduces *aspects*. An aspect is the modularization of a crosscutting concern. Two approaches are distinguished in AOP. Asymmetric approaches consider aspects as different entities from those ones that compose the base system (objects or components), such as AspectJ [25], or JAsCo [44]. Symmetric approaches 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 in a symmetric way aspect-oriented and component-oriented approaches to benefit from the modular properties of both approaches, such as Fractal-AOP [14] or FAC [39].

In Scl, we adopt a symmetric approach with limited aspect-oriented features which are provided through special connectors and ports characteristics. The *join points* – well defined points in the execution of a program where aspects can be woven – are generally method calls, method call receptions, method executions or attribute accesses. The supported joint points in Scl are: before/after/around a service invocation or connection/disconnection on a port. Figure 11 shows an example that uses an SclFlowConnector and a regular Logger component to add the logging support to a component c through its port `pc`.

In a SclFlowConnector, all source ports are coupled with a keyword (beforeServiceInvocation, beforeConnection, ...) that specifies when the glue code has to be executed. At execution time, when a service invocation arrives on a port, glue code of attached connectors are executed in the same order as in AOP (around, before, after). Conflicts are possible, for example if multiple glue codes have to be executed before a service invocation on the same port, the glue code of the last connected connector will be executed first. This rule lets the architect deal with potential weaving problems.

```
SCLComponentClassBuilder createComposite: #C.

C>>init
  self addSubComponent: A new named: a.
  self addSubComponent: B new named: b.
  self forwardPort: (b port: #pb).

  SclBinaryConnector new
    source: (b port: #rb)
    target: (a port: #pa) ;
    connect.
```

Fig. 8. Declaration of a composite component class

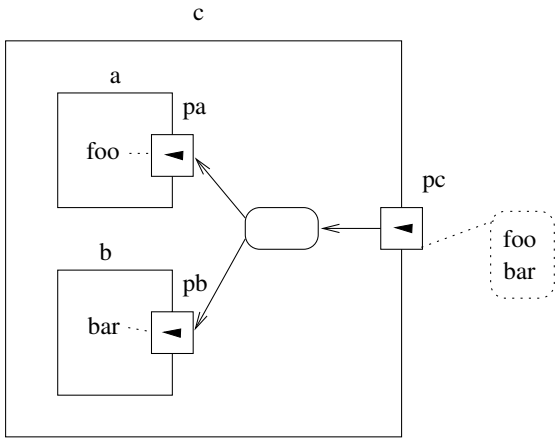

Fig. 9. Port forwarding using a connector

```
SCLComponentClassBuilder createComposite: #C.

C>>init
  self addSubComponent: #A named: a.
  self addSubComponent: #B named: b.
  self addPort: (SclPort new: #pc
                 provides: (SclInterface new with: {#foo. #bar}).

  SclForwardConnector new
    sources:{self port: #pc}
    targets:(b port: #pb).
       (a port: #pa)}
    glue: [ :sources :targets :message |
      (message selector == #foo) ifTrue:[
        ^targets first perform: message
      ] ifFalse: [
        (message selector == #bar) ifTrue:[
          ^targets second perform: message
        ].
      ].
    ];
    connect.
```

Fig. 10. Using a connector to forward services in a composite component

```
l := Logger new.

SclFlowBinaryConnector new
  source:((c port: #pc) beforeServiceInvocation)
  target:(l port: #Logging)
  glue: [ :source :target :message |
      target log: 'The ', message selector, ' message will be sent to a'
  ];
  connect.
```

Fig. 11. Modify the control flow using a connector

3.5 Component Properties and Publish/Subscribe Connections

Triggering operations as a consequence of state changes in a component is related to Observer design pattern [17] or *procedural attachments* [35]. 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 [27]) and "models". More generally, the publish/subscribe [13] communication protocol is a very useful communication pattern to decouple software entities as said in [18]: *"The main invariant in this style is 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 *Java Bean* model, the programmer has to manage explicitly the subscribers list (add and remove subscriber methods). In the CCM (Corba Component Model), 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 can not 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 order to increase the component reuse, we have to decouple the connection code from the business code written by the component programmer. The programmer has to focus on the business code and the design of the component i.e what it requires and what it provides. In Scl, there are three 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 using regular SclCallConnector to link publishers and subscribers.
2. The component programmer does not integrate the event signaling in the component code and SclFlowConnectors can be used by the architect to detect the events that he needs. For example, if the architect wants to detect when a 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.
3. The component programmer has declared *properties*. This property concept enhances the idea of property of the Javabeans component model [22] with strict separation between component code and connection code. A property is an external state of a component. 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. Figure 12 shows the Scl code for this declaration.

```
SCLComponentClassBuilder create: #Counter.

Counter>>init
  self addAttribute: #value.
  self addPort: (SclPort new: #Counting
          provides: (SclInterface new with: {#dec. #inc}).
  self addProperty: #Count read: [^value] write: [ :nv | value := nv ].

C>>inc
  self count: (self count + 1)

C>>dec
  self count: (self count - 1)
```

Fig. 12. A Counter component class with a property

When a programmer declares a property, the component is automatically composed of 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. 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

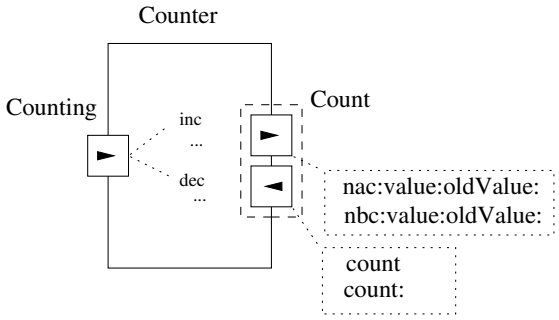

Fig. 13. A counter component with a value property

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

SclBinaryNACConnector new
  source: ( counter notifyPortOf: #Count)
  target: ( gui port: #Displaying )
  glue: [ :source :gui :message |
    gui displayText: (message arguments second).
  ]; connect.
```

Fig. 14. A state changes connection based on a component property

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 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). Special or regular connectors can be used to connect properties since they are just two regular ports. An example of connection using properties is depicted on Figure 13 and the corresponding Scl code is shown on Figure 14.

In this example, a SclBinaryNACConnector is used. 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). Actually, Scl provides different kinds of connectors like SclBinaryNACConnector, SclBinaryNBCConnector, 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.

4 Implementation

The actual prototype of Scl [28] is implemented in Squeak/Smalltalk [24]. Squeak is an open and highly portable implementation based on the original Smalltalk-80 system [19]. Figure 15 shows a part of the class diagram of the core model.

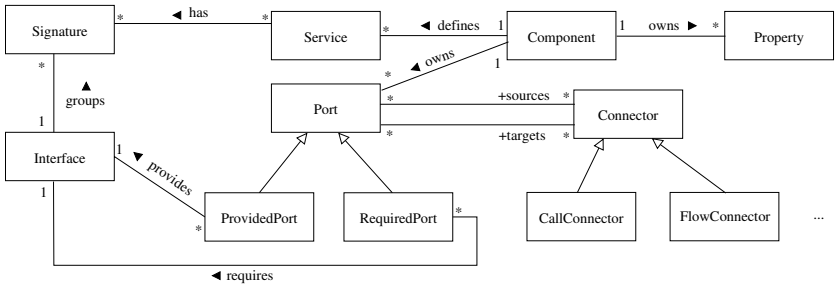

Fig. 15. UML class diagram of the current implementation of Scl

This figure shows only main connector families. In [33], a taxonomy of software connectors has been established and eight kinds of connectors have been identified. Similarly to the work done for ArchJava in [3], we have implemented connectors of each kind. This shows that the connector model of Scl is suitable to perform a large variety of connectors.

The current Scl syntax is the same as the Smalltalk one although some changes in semantics have been done. For example, the syntactical receiver of a service invocation is a port but the real Smalltalk receiver is not this port but the component which this port belongs to. Because we do not implement Scl with an evaluator or a compiler but directly with Smalltalk constructs, it is easier to change and evolve the implementation. It is also difficult to implement special things that are too far from the Smalltalk mechanisms.

5 Related Work

Understanding or teaching COP. Picolo [30] and BoxScript [29] are two frameworks for introducing (teaching) components. Picolo is written in Python and Boxscript in Java. They are small and contrary to Scl, they integrate a simple binary connection mechanism.

Architecture description languages (ADLs). These languages are an important part of the current researches in component-oriented languages. In [32], a classification of the most known ADLs has been established. For example, Wright [5] is one of these languages that integrates connector support. But Wright, as many of these languages, is dedicated to simulation and formal verification. Since ADLs are not executable languages, it is not possible to build an application using it.

ArchJava [2,3] is a Java extension introducing software architecture concepts to express architectural structure of applications within an implementation in order to ensure that the implementation conforms to architectural constraints. ArchJava classes support bidirectional ports in which methods are provided, required or broadcasted. The primitive connection mechanism (connect keyword) is a coarse-grained one because it is based on bidirectional ports. ArchJava does not support properties and component programmers have to write code in components to enable connections based on component state notifications.

Fractal [8] is a recursive and reflective component model. A component has external interfaces (ports) which provides (server interface) or required (client interface) a defined set of services. Components are connected through bindings between external interfaces. A primitive binding is a fixed interface connection mechanism that binds one client interface with one server interface. Binding components also called connectors represent composite bindings to create complex connections. In Scl, components and connectors are different concepts because these two concepts fulfill different purposes, components are the business reusable software units while connectors have to fix connection semantics and deal with connection problems. Julia is the implementation reference of the Fractal model in Java, and Fractalk [15] is an implementation of Fractal in Squeak.

ComponentJ [42] is another Java extension for component-oriented programming. Components provide or require one interface per port. The component programmer defines methods inside method blocks that can be plugged into ports. Plug operations bind one component method block or port to a port according to their interfaces. Component composition is done through dynamic composition (`compose` keyword) and returns a new component. ComponentJ is a strongly typed language ensuring plug operations and composition. There is no connector support in ComponentJ and it is only possible to connect components inside a composite. A component can only be instantiated if it is closed (without unbounded required services) even if all required services are not necessary for the current application.

Javabeans [22] has been one of the first component models allowing programmers to connect independently developed software entities. Javabeans programmers have to write special connection code (essentially Observable code from the Observer pattern) and to respect syntactical rules to ensure that their Javabeans can be automatically connected with other Javabeans using automatic Adaptor [17] generation. Our properties, inspired from the Javabeans model, do not enforce component programmers to write specific connection code.

Component-oriented languages. These languages enable to code an application using a component approach. These languages do not integrate the ADLs features and provide object-oriented extensions to program with components. For example, Lagoona [16] is based on the idea that components are modules that contain class and message definitions. Note that most of proposed component-oriented languages are Java extensions (Lagoona, Keris, ComponentJ, ArchJava, Javabeans, Julia/Fractal, ...). There are few proposals using a dynamic language

and none in Smalltalk, except Fractalk that is an implementation of Fractal in Squeak.

Mixing component and aspect oriented programming. As said in section 3.4, we only consider symmetrical approaches such as FAC [39] or FractalAOP [14] where aspects are regular components. This is to increase the reuse of components that can be used as regular components as well as aspects 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 a base component or as an aspect component and uses the special connector SclFlowConnector. This Scl feature is clearly not a complete support of AOP, but an attempt to bring some flexibility of AOP in Scl respecting that components are independently developed and composed.

6 Conclusion

Component-based software development is founded on the unanticipated composition of independently developed software components. Such a mechanism must be offered to programmers and many languages integrate some concepts and mechanisms to achieve this. In this paper, we present Scl a concrete component-oriented language. We believe that Scl represents a simple and uniform synthesis of current proposal on component-oriented programming. Scl also brings new features like a general purpose connector model. Connectors are useful to provide an extensible connection mechanism that solves component connection problems. They offer a unified entity that enable standard required/provided connections and also event-based component connections due to special connectors and component properties. A component programmer only declares properties that represent external state of components. A software architect can express connections on the basis of properties notifications with the same connection mechanism based on connectors.

Ongoing researches on Scl are focused on three areas. First, extending the component model of Scl in order to better support dynamically features. For example, dynamically adding or removing ports to a component could be a great solution to deal with components that have a potentially unbounded number of connections such as a Web Server component. Second, we plan to provide a stable release of the current implementation of Scl and integrate tools dedicated to component-oriented programming. And finally, developing large scale applications using Scl will certainly show us interesting results about the Scl expressiveness compared to existing component-oriented languages which are mainly statically typed ones.

References

1. Franz Achermann and Oscar Nierstrasz. Applications = Components + Scripts – A Tour of Piccola. In Mehmet Akşit, editor, *Software Architectures and Component Technology*, pages 261–292. Kluwer, 2001.

2. Jonathan Aldrich, Craig Chambers, and David Notkin. ArchJava: Connecting Software Architecture to Implementation. In *ICSE*, pages 187–197. ACM, 2002.
3. Jonathan Aldrich, Vibha Sazawal, Craig Chambers, and David Notkin. Language support for connector abstractions. In Luca Cardelli, editor, *ECOOP*, volume 2743 of *Lecture Notes in Computer Science*, pages 74–102. Springer, 2003.
4. Robert Allen. *A Formal Approach to Software Architecture*. PhD thesis, Carnegie Mellon, School of Computer Science, January 1997. Issued as CMU Technical Report CMU-CS-97-144.
5. Robert Allen and David Garlan. The Wright Architectural Specification Language. Technical report, School of Computer Science, Carnegie Mellon University, Pittsburgh, 1996.
6. Dusan Balek and Frantisek Plasil. Software connectors and their role in component deployment. In *Proceedings of DAIS'01*, Krakow, Poland, September 2001. Kluwer Academic Publishers.
7. Manfred Broy, Anton Deimel, Juergen Henn, Kai Koskimies, Frantisek Plasil, Gustav Pomberger, Wolfgang Pree, Michael Stal, and Clemens A. Szyperski. What characterizes a (software) component? *Software - Concepts and Tools*, 19(1):49–56, 1998.
8. Eric Bruneton, Thierry Coupaye, Matthieu Leclercq, Vivien Quéma, and Jean-Bernard Stefani. An Open Component Model and Its Support in Java. In Ivica Crnkovic, Judith A. Stafford, Heinz W. Schmidt, and Kurt C. Wallnau, editors, *CBSE*, volume 3054 of *Lecture Notes in Computer Science*, pages 7–22. Springer, 2004.
9. Martin Büchi and Wolfgang Weck. Compound types for Java. In *OOPSLA'98: Proceedings of the 13th ACM SIGPLAN conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 362–373, New York, NY, USA, 1998. ACM Press.
10. Luca Cardelli. *The Handbook of Computer Science and Engineering*, chapter 103, Type Systems, pages 2208–2236. CRC Press, Boca Raton, FL, 1997.
11. John Cheesman and John Daniels. *UML components: a simple process for specifying component-based software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2000.
12. Michael Eichberg. Mda and programming languages. In *Workshop on Generative Techniques in the context of Model Driven Architecture (OOPSLA '02)*, 2002.
13. Patrick Th. Eugster, Pascal A. Felber, Rachid Guerraoui, and Anne-Marie Kermarrec. The many faces of publish/subscribe. *ACM Comput. Surv.*, 35(2):114–131, 2003.
14. Houssam Fakih, Noury Bouraqadi, and Laurence Duchien. Aspects and software components: A case study of the FRACTAL component model. In Minhuan Huang, Hong Mei, and Jianjun Zhao, editors, *International Workshop on Aspect-Oriented Software Development (WAOSD 2004)*, September 2004.
15. FracTalk. Fractal Components in Smalltalk `http://csl.ensm-douai.fr/FracTalk`.
16. Peter H. Fröhlich, Andreas Gal, and Michael Franz. Supporting software composition at the programming-language level. *Science of Computer Programming, Special Issue on New Software Composition Concept*, 56(1-2):41–57, April 2005.
17. Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, March 1995.
18. David Garlan and Mary Shaw. An introduction to software architecture. In V. Ambriola and G. Tortora, editors, *Advances in Software Engineering and Knowledge Engineering*, pages 1–39, Singapore, 1993. World Scientific Publishing Company.

19. Adele Goldberg and David Robson. *Smalltalk-80: The Language*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1989.
20. Bernhard Gröne, Andreas Knöpfel, and Peter Tabeling. Component vs. component: Why we need more than one definition. In *ECBS*, pages 550–552. IEEE Computer Society, 2005.
21. Object Management Group. Uml 2.0 superstructure specification. Technical report, Object Management Group, 2004.
22. Graham Hamilton. JavaBeans. API Specification, Sun Microsystems, July 1997. Version 1.01.
23. George T. Heineman and William T. Councill, editors. *Component-based software engineering: putting the pieces together*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001.
24. Dan Ingalls, Ted Kaehler, John Maloney, Scott Wallace, and Alan Kay. Back to the future: the story of Squeak, a practical Smalltalk written in itself. In *OOPSLA '97: Proceedings of the 12th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 318–326, New York, NY, USA, 1997. ACM Press.
25. Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An Overview of AspectJ. In Jørgen Lindskov Knudsen, editor, *ECOOP*, volume 2072 of *Lecture Notes in Computer Science*, pages 327–353. Springer, 2001.
26. Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In Mehmet Akşit and Satoshi Matsuoka, editors, *11th Europeen Conf. Object-Oriented Programming*, volume 1241 of *LNCS*, pages 220–242. Springer Verlag, 1997.
27. Glenn E. Krasner and Stephen T. Pope. A cookbook for using the model-view-controller user interface paradigm in smalltalk-80. In *Journal of Object-Oriented Programming*, volume 1, pages 26–49, Août-Septembre 1988.
28. Simple Component Language. `http://www.lirmm.fr/~fabresse/scl/`.
29. Y. Liu and H. C. Cunningham. Boxscript: A component-oriented language for teaching. In *43rd ACM-Southeast Conference*, volume 1, pages 349–354, March 2005.
30. Raphaël Marvie. Picolo: A simple python framework for introducing component principles. In *Euro Python Conference 2005*, Göteborg, Sweden, june 2005.
31. M. D. McIlroy. Mass produced software components. In P. Naur and B. Randell, editors, *Proceedings, NATO Conference on Software Engineering*, Garmisch, Germany, October 1968.
32. Nenad Medvidovic and Richard N. Taylor. A classification and comparison framework for software architecture description languages. *Software Engineering*, 26(1):70–93, 2000.
33. Nikunj R. Mehta, Nenad Medvidovic, and Sandeep Phadke. Towards a taxonomy of software connectors. In *ICSE '00: Proceedings of the 22nd international conference on Software engineering*, pages 178–187, New York, NY, USA, 2000. ACM Press.
34. Microsoft. DCOM technical overview. Microsoft Windows NT Server white paper, Microsoft Corporation, 1996.
35. M. Minsky. A Framework for Representing Knowledge. In P. Winston, editor, *The Psychology of Computer Vision*, pages 211–281. mgh, ny, 1975.
36. Oscar Nierstrasz and Laurent Dami. Component-oriented software technology. In Oscar Nierstrasz and Dennis Tsichritzis, editors, *Object-Oriented Software Composition*, pages 3–28. Prentice-Hall, 1995.

37. Object Management Group. *Model Driven Architecture*, 2003. *http://www.omg.org/mda*.
38. D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Comm. ACM*, 15(12):1053–1058, December 1972.
39. N. Pessemier, L. Seinturier, L. Duchien, and T. Coupaye. A model for developing component-based and aspect-oriented systems. In *Proceedings of the 5th International Symposium on Software Composition (SC'06)*, volume 4089 of *Lecture Notes in Computer Science*. Springer, March 2006.
40. Frantisek Plasil and Stanislav Visnovsky. Behavior protocols for software components. *IEEE Trans. Softw. Eng.*, 28(11):1056–1076, 2002.
41. Johannes Sametinger. *Software engineering with reusable components*. Springer-Verlag New York, Inc., New York, NY, USA, 1997.
42. João Costa Seco and Luís Caires. A basic model of typed components. *Lecture Notes in Computer Science*, 1850:108–129, 2000.
43. Mary Shaw. 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*, pages 17–32, London, UK, 1996. Springer-Verlag.
44. Davy Suvée, Wim Vanderperren, and Viviane Jonckers. Jasco: an aspect-oriented approach tailored for component based software development. In *AOSD '03: Proceedings of the 2nd international conference on Aspect-oriented software development*, pages 21–29, New York, NY, USA, 2003. ACM Press.
45. C. Szyperski. *Component Software: Beyond Object-Oriented Programming (2nd Edition)*. Addison-Wesley, 2002.
46. Rob C. van Ommering. Koala, a component model for consumer electronics product software. In Frank van der Linden, editor, *ESPRIT ARES Workshop*, volume 1429 of *Lecture Notes in Computer Science*, pages 76–86. Springer, 1998.
47. Matthias Zenger. Type-safe prototype-based component evolution. In *Proceedings of the European Conference on Object-Oriented Programming*, Malaga, Spain, June 2002.
48. Matthias Zenger. Keris: evolving software with extensible modules: Research articles. *J. Softw. Maint. Evol.*, 17(5):333–362, 2005.