CLIC: a Component Model Symbiotic with Smalltalk

Noury Bouraqadi – Luc Fabresse

Université Lille Nord de France
Ecole des Mines de Douai
France
{bouraqadi,fabresse}@ensm-douai.fr – http://vst.ensm-douai.fr/Clic

Abstract

Evolving object-oriented code such as replacing a part of a system, is not always as easy as it should be. This is because object-oriented languages do not enforce code modularisation. Component-oriented approches target this issue by expliciting software architecture in terms of components and connections between them. However, there is little language support for component-oriented programming. Existing attempts are not really well integrated with the hosting object-oriented language. In this paper, we present the CLIC component model, its integration and its support in the Smalltalk language.

Keywords Component Model, Component-Oriented Language

1. Introduction

Component approches are a promising track to build well modularized and easily evolvable software out of reusable components and connections. Programming component-based applications is currently carried out using object-oriented languages which do not offer specific abstractions to ease component-oriented programming (COP) and have to be used in a disciplined way to guarantee a COP style. By disciplined way, we mean that design patterns and/or syntactic conventions have to be used scrupulously in the code if we want to benefit the component-oriented advantages. A component-oriented language (COL) eliminates these burdens because it offers high-level component-oriented abstractions and mechanisms that can be directly used by programmers. Even in a model tranformation process where the code is generally partially generated, using a COL is desirable because the code generation will be easier and the resulting code will be more readable and easier to be completed by a programmer.

However, there are few COL. Existing ones are generally not well integrated with the hosting (often object-oriented) language. For instance, the different component models builts on top of Smalltlalk are not well integrated with the Smalltalk language and they provide only limited COP support. In this paper, we propose to to address this limitations. We present CLIC, a component-oriented language specifically designed to be symbiotic with the Smalltalk language.

In the following, we first set up the vocabulary that will be used throughout the paper and provide a set of definitions related to the concept of component in section 2. Next, section 3 is dedicated to the state of the art. We provide an overview of existing component models implemented on top of Smalltalk, and discuss their limitations. Then, we introduce the CLIC component model in section 4. We present the design decisions behind CLIC and provide insight about their implementation. Section 5, concludes the paper and draws some future works.

2. Components Related Concepts and Definitions

Component approches have brought many new or adapted abstractions and mechanisms such as *component*, *module*, *connection*, *composition*, *binding*, *port*, *interface*, *connector*, *service*, *message* but their interpretation vary quite widely from one proposal to another. In this section, we define the concepts that we will use in this paper.

2.1 Component's Structure

A *component* [Szyperski 1998] is a software entity that can be deployed independantly and dynamically assembled with other components. A component is created by instantiating a *component descriptor* which describes its structure (*ports* and *attributes*) and behavior (*operations*).

A component has *ports*. All interactions with a component have to pass through one of its ports. Unidirectionnal models distinguish two kind of ports: *provided ports* and *required ports*. A component provides *operations* (same as

International Workshop on Smalltalk Technlogies Sept 2009, Brest, France.

OOP methods) through a provided port and uses operations of other components through its required ports. Bidirectionnal models do not make this distinction and operations can be provided or required through the same port. A port has some contracts attached to. There are different kinds of contracts such as syntactic contracts which describe the signatures of the operations provided or required through the port. More complex contracts can be attached to a port such as protocols (a description of the valid operation call sequences) or quality of service requirements.

Attributes are configurable variables local to a component. An attribute is materialized by a couple of accessing operations (getter and setter) provided through one port of the component. In order to configure components for purpose of reuse in different application contexts, the default values of attributes should be easy to change. These default values will be used at instantiation-time to automatically configure newly created components.

2.2 Connection, Composite and Architecture

The *architecture* of a software system defines the structure of the system in terms of components and of *connections* among those components [Shaw et al. 1995]. Components manage business functions while connections are dedicated to components coordination and interaction. One of the most basic connection mechanisms is the direct *binding* of required ports to provided ports. Complex connection support is provided when connections are reified as *connectors* [Shaw 1996]. A connector also links components together but it may hide wider complexity such as communication between remote components or constraints checking (integrity, coordination, security, robustness, ...). A taxonomy of sotware connectors has been identified [Mehta et al. 2000] and predefined software connectors ease components assembly.

The architecture of large scale software must be viewed at different levels of details in order to present a macroscopic or a microscopic view of the software. *Hierarchical* component models enable viewing an architecture at different levels of details because they support *composites*. A composite is a component that abstracts an architecture i.e encapsultes some components – called *subcomponents* – and their connections. Subcomponents may be connected to each others and their ports may be exported and made available as composite ports. A subcomponent may be itself a composite. Then, reusing a composite is just like reusing an assembly.

2.3 Summary

Figure 1 summarizes the main component related concepts presented in this section.

3. Existing Component Models for Smalltalk

We give in this section an overview of existing component models implemented on top of Smalltalk, and discuss their limitations.

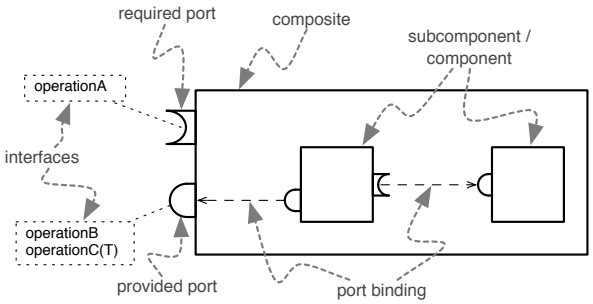

Figure 1. Overview of the components related concepts

3.1 SCL

SCL [Fabresse et al. 2008] (Simple Component Language) is a component-oriented language designed to support and ease component-oriented programming (COP). SCL is dedicated to COP because it enforces:

- *decoupling* means that a component never contains a reference to another component,
- *unanticipation* means that the concrete components that will be actually connected on to one component are unknown by its programmer.

In SCL, a component is a runtime entity created by the instantiation of a descriptor. A component has ports. A port is decribed by one interface that declares a set of operation signatures. A component provides its operations through its provided ports and invokes the operations of other components through its required ports. One of the key feature of SCL is that operation invocation must be sent through a required port. This implies that the programmer doesn't know which operation of which component will be executed at runtime. It depends on the port bindings of the required port. More sophisticated bindings can be defined using connectors and glue code. A connector is a component that receives operation invocations through its provided ports and transmits them throught its required ports by executing the glue code. The glue code is modularized in a special operation attached to a provided port. This operation is executed each time a operation invocation arrived through the attached provided port. Thanks to syntactic sugar, glue code can be written as a lexical closure which does not burden programmers with the writing of descriptors for simple connectors. The extensible and uniform component connection mechanism in SCL is based on ports, bindings, glue code and connectors. This mechanism addresses the key issue of the unanticipated connection of independently developed components.

SCL also integrates unparalleled features. Basic types (numbers, characters, ...) are presented to developpers as regular components equipped with one provided port. This uniformity allows basic types to be treated using the same mechanisms as those for regular components. In SCL, argu-

ments passing is achieved using automatic connection and disconnection of effective arguments (components) with the receiver component. This innovative mechanism prevents references escape and therefore communications outside of connections defined in the archietcture description. Autoreferences (a component invokes one of its provided operations) are achieved in SCL using regular operation invocations through the `self` port. All components have an internal required port named `self` dedicated to autoreferences. This special port unifies autoreferences and other operation invocations. In the area of separation of concerns, SCL enables the use of a component as a "regular" component or a "crosscutting" component (as in aspect-oriented programming) thanks to a black-box joint point model and additionnal binding primitives.

SCL is a language currently prototyped on top of Smalltalk. In the current prototype implementation, all SCL concepts have been reified. Error handling (`doesNotUnderstand:`) coupled with message sending reification [Ducasse 1999] are used to implement the SCL mechanisms. Although SCL has currently a Smalltalk compliant syntax, it is not a Smalltalk exetension. A SCL programmer should not use Smalltalk's mechanisms such as inheritance (SCL advocates reuse by connection only), message sending (SCL relies on operation invocation and specific arguments passing), ... All of this makes SCL clearly not symbiotic with Smalltalk.

3.2 FracTalk

FracTalk [1] is a Smalltalk implementation of the Fractal hierarchical component model [Bruneton et al. 2002]. We first start by presenting the Fractal model. Then, we provide a description of the FracTalk implementation and how it is integrated with Smalltalk.

Fractal specifies a component as a run-time entity that can be handled through one of its provided ports (named *server interfaces* in Fractal). A component can also have an arbitrary number of required ports (named *client interfaces*) that support connections to ports provided by other components.

A required port can be optional or not. A required port op is optional if the component functioning is not altered if op is not connected.

A required port can be singleton or collection. A required port is singleton if it can not be connected to more that one provided port. A required port is collection if it can be connected to several provided ports. Actually, a collection port is handled as a dictionary of required ports.

The ports can be either functional or non-functional. Functional ports refer to application related functionality. A component receives invocation of its provided operations through its provided functional ports. It sends invocations of operations provided by other components through its required functional ports. Provided functional ports allow invoking operations implemented by the component. The actual implementation of a component's functional part is called *content*. It refers both to the implementation of provided operations as well as to the component's *attributes*. This implementation can be plate-forme independent for primitive objects. Conversely, the content of a composite component is a set of components.

Non-functional ports (named *controllers*) allow managing the component and support reflection. The Fractal designers state that the set of non-functional ports is open, that is, designers can add new non-functional ports. Several non-functional ports were described in the Fractal specification. They provide a variety of operations such as setting up the component attributes, connecting the component required ports, managing the component life-cycle (e.g. start, stop), and introspecting the component (e.g. retrieving all its ports)

The full set of ports (provided and required, functional and non-functional) of every Fractal component is part of the component's *membrane* which implements non-functional features such as the component life-cycle or connection handling. The membrane is the outer most part of the component and encapsulates the content. Indeed, it intercepts all incoming operation calls that have to be processed by the content. It also intercepts all outgoing operation calls sent by the content.

Regarding component descriptions, the Fractal specification mainly focuses on the membrane description. Developers have to clearly declare all ports that are provided or required by a component. The actual implementation is leaved to the platform implementors which must rely on the Factory design pattern.

In FracTalk, the content of a primitive component is implemented as a plain object. Also, every port is implemented as a single object in order to ensure that every port allow invoking only declared operations. Therefore, a single component materializes as multiple objets. Besides, the description of a component is scattered over multiple classes.

Another limitation of FracTalk is the the difficulty to make use of Smalltalk libraries. Smalltalk objects aren't full fledged components since they do not have a membrane and then does not provide expected non-functional ports. Therefore, the only mean to use a Smalltalk object in a FracTalk application is to encapsulate it in the content of some component.

4. CLIC

We present here design choices made in CLIC[2] and explain the rational behind. To design CLIC, we started from Smalltalk and extended it with component features. Our goal was to get a simple model that is tightly coupled to Smalltalk, while compliant with the concept of component

[1] `http://vst.ensm-douai.fr/FracTalk`

[2] "Clic" in french is an onomatopoeia that mimics the sharp sound of two objects coming quickly into contact, like building parts that get connected.

as defined by Szyperski [Szyperski 1998]. The symbiosis with Smalltalk allows us to avoid many performance losses, and benefit from Smalltalk dynamicity.

4.1 CLIC Components

A CLIC component is a run-time entity with explicit dependencies, characterized by:

- *1 provided port.* The unique provided port is the only mean to interact with a CLIC component. This port gives access to provided operations which are messages understood by the component.
- *0 or more required ports.* Required ports hold references to other components. A required port can be simple, and support connections to only one component each time. Conversely, a required port can be complex, and bear connections to a collection of components. A component can function correctly if all its required simple ports are connected to other components.
- *0 or more attributes.* A component can be a composite, that is it can encapsulate some subcomponents. Subcomponents are referenced as attributes. They can be either private to a single component or shared among two or more components. For every attribute, the component support accessing operations named after the attribute. Also, for each attribute, the component exhibits a complex required port that fires events when the attributes values change.
- *1 architecture.* The set of connections among attributes are made explicit as an *architecture*. This architecture is used on component creation to actually set up connections. When a subcomponent is replaced, the architecture allows to identify connections to destroy in order to free the obsolete subcomponent, and connections to build in order to connect the new subcomponent.

4.1.1 *Provided Port*

In constrast to components in other models, each CLIC component provides a single port. This port is the only mean to interact with a CLIC component, since it gives access to all operations provided by the component. Indeed, we believe that this approach favors more reuse. In component models supporting multiple provided ports per component, it is not always easy to split component functionality over multiple ports. Indeed, designers of any C1 component don't know beforehand which operations will be specified by each required port of client components. For example, designers of C1 may decide to make two operations op1 and op2 available through two different provided ports $\mathsf{proP1}_{C1}$ and $\mathsf{proP2}_{C1}$. But, a client component C2 may call the two operations op1 and op2 through a single required port reqP_{C2}. Therefore, there is a structural mismatch that forbids connections between C1 and C2, even though C1 and C2 match semantically.

We believe that splitting provided operations over different provided ports should be avoided. Still the information about relationships between provided operations is an interesting information to have for a component client. Therefore, while we enforce a single port in CLIC, we rely on method categories borrowed from Smalltalk to provide a first level of information about roles operation and relationships between operations.

4.1.2 *Required ports*

A CLIC component can have a set of required ports in order to get connected and then interact with other components. A required port in CLIC can be of kind singleton or collection. The novelty of CLIC is that component developers are offered a variety of collection required ports borrowed from the Smalltalk collection hierarchy (Set, Array, Dictionary...). When implementing a component description, developers choose which one to use. This choice is made on development time, because the interaction with components connected through collection ports is handled the way Smalltalk deals with collections. So, developers need to know which kind of collection they are dealing with to use the right protocols (e.g. keysAndValuesDo: for dictionaries, indexOf: for arrays and ordered collections).

4.1.3 *Attributes*

Existing component models such as Fractal do make a distinction between attributes and subcomponents. This is because often attributes are primitive types and thus treated differently. Nevertheless, in CLIC we have a unified environment where every entity is a component, since all Smalltalk objects are components. Therefore, we merged the concepts of attributes and subcomponents[3]. Indeed, we believe that the distinction between the two concepts only increases the complexity of the component models without bringing any significant added value.

However, we made explicit the concept of shared attributes that is either forbidden or implicit in other models such as Fractal. A shared attribute is typically a global component (i.e. referenced in the system dictionary) and that is used in the implementation of different components. We belive that this concept is as useful as class variables.

Last, in order to favor reuse and unanticipated evolution, we decided to include a built-in support for the Observer design pattern. Therefore, the component has a required port of kind collection for every attribute. When an attribute changes, an event is fired through the corresponding required port. The event object provide information about the changed component, the changed attribute, and the new and the old attribute values.

[3] We do use the two words interchangeably in the following.

4.2 CLIC Classes

In CLIC, component descriptors are classes. Every CLIC component is an instance of a class. The class defines the component characteristics exposed in section 4.1. Particularly, it hold methods that implement operations that can be called through the component's unique provided port. In order to ease reuse and evolution, CLIC ensures that all dependencies are explicit. All required globals (e.g. other classes) referenced in the class definition and methods can be automatically retrieved by sending the dependencies message to the class.

A CLIC class definition (see figure 2) also declares private and shared attributes. This declaration results into two extra provided operations for accessing each attribute: one for reading and the other for writing the attribute. In the provided example, every counter will respond to messages count and count: although the developers did not implemented these methods.

The class definition also includes initialization directions for declared or inherited attributes. That is a reference to a literal or a global object (stored in the Smalltalk system dictionary). In the provided counter class definition, the initial value of the count attribute is the literal 0. It is possible to a express a unary message which receiver is a global. The message will be valued on instantiation time and answer will be the initial value of the referenced attribute. For example, the directive `#myAttribute -> Time @ now` will initialize the attribute myAttribute with the result of the message now sent to the Time class.

```
(CLComponent subclass: #CLCounter
    localPrivateAttributeNames: #(count)
    privateAttributesInitDict: {
        #count -> 0}
    sharedAttributeNames: #()
    sharedAttributesInitDict: {}
    localRequiredPortsDict: {}
    category: #'ClicExamples-Clock')

        architectureFrom: {};
        exportedPrivateAttributesDictFrom: {};
        operationsExportDictFrom: {};
        operationsDefaultReceivers: #();
        exportedRequiredPortsDictFrom: {}
```

Figure 2. Definition of a counter component class in CLIC

The definition of a class includes the declaration of required ports. Names of each required port and its kind are provided. The port kind can be either a CLSingletonPort, or any collection class (Array, Dictionary, Set...). For example, a class definition may include the following declaration of required ports:

```
requiredInterfacesDict: {
    #myRequiredPort -> CLSingletonPort.
    #myOtherRequiredPort -> Set}
```

It states that instances of this class are component that have two required ports. The first port, named myRequiredPort is a singleton, and as such it can reference only a single component. The second port, named myOtherRequiredPort is of kind Set and thus, it can reference multiple non-equal components.

The second part of the class definition is provided as a cascade of messages sent to the component class. First, it states the component architecture, that is how attributes are connected. The directive below states that the port requiredPort of the sub-component referenced as someAttribute will be connected to the component referenced by attribute otherAttribute.

```
(#attrib1 @ #port) => #attrib2.
```

When a subcomponent holds a collection port that needs to be connected to more than one other subcomponent, the last part of the directive is replaced by an array, as expressed below:

```
(#attrib1 @ #port) => #(attrib2 attrib3).
```

Besides the architecture, the second part of the definition expresses exports. This a facility avoids writing glue code to make features of subcomponents available from the outside. Thus, some operations actually implemented in a subcomponent can made available, so the component can display them on its provided port. When called, the operation is executed by the subcomponent.

Listing explicitly provided operations to export can be tedious, we introduced the operationsDefaultReceivers facility to export all provided operations. It is simply an ordered list of attributes. When a component receives a message that refers neither to operations it implements, nor to exported operations, the operation is looked up in the defaults receivers list. The first subcomponent that implements the operation will process message.

A class definition can also include required ports exports. Such exports are useful when designers decide to keep some subcomponents required ports unconnected. This is done often for sake of genericity. Exporting a subcomponent required ports means that it is the responsibility of the component assembler to decide wich component to use for performing the needed functionalities.

Export is also possible for sub-component attributes. When an attribute is exported, its accessors become available operations. Also, the attribute observers required port becomes available. So, outer components can be connected in order to react to the attribute changes.

CLIC inheritance model is analogous to Smalltalk: each class has a single superclass. A CLIC class inherits every part of the definition from its superclass. A subclass can override attribute initialization directives and extend the other features (attribute declarations, required ports, architecture...).

4.3 CLIC and Smalltalk Symbiosis

A major noticeable difference between CLIC and other component models is that CLIC relies heavily on the Smalltalk language feature and the Smalltalk virtual machine. First of all each component is implemented exactly as a single object. Provided ports are but object references, and required ports are instance variables. This design decision contrast with other approaches where a component materializes as a set of objects, including objects that act as reifications of ports.

A nice consequence of this design decision, is that the use of CLIC introduces no overhead on application memory footprints. Another interesting consequence is that every Smalltalk object is also a component, so we get a fully component-based programming environment and libraries. Indeed, a Smalltalk object does exhibit a single provided port exactly as CLIC components do. Still, we consider plain Smalltalk objects as "dirty" components since they are tightly coupled to each others and their architecture is hidden.

Regarding the class level, while CLIC is a full fledged component model, the difference between CLIC classes and plain Smalltalk classes is structural. Indeed, CLIC code is compiled using the Smalltalk compiler. We rely on Smalltalk reflective capabilities, namely metaclasses, to introduce CLIC classes. But, the core behavior of CLIC classes is fully compliant with Smalltalk classes, so the Smalltalk virtual machine treats CLIC classes as plain Smalltalk classes. CLIC adheres to single inheritance, and operations are implemented as methods stored in the method dictionary. Therefore, CLIC introduces no overhead when processing operations. Operation calls are but messages sends that are bounds to methods by the virtual machine by simply looking them up into method dictionaries.

Last, provided ports of kind collection also reflect the symbiosis between CLIC and Smalltalk. Indeed, the declaration of a collection port refers to one of the Smalltalk collection hierarchy classes. It means that connections through a collection port are references hold by an instance of the provided class. Thus, CLIC developers fully benefit form of the rich Smalltalk collection hierarchy.

5. Conclusion and Future Work

In this paper we presented the CLIC component model and its Smalltalk-based implementation. CLIC was thought from the beginning as an extension of Smalltalk to support full-fledged components. We show indeed, that CLIC provides component features such as ports, attributes, or architecture.

From the implementation point of view, we fully relied on Smalltalk reflective capabilities. Thus, from the Smalltalk virtual machine point of view, CLIC components are but objects and their descriptors are extended Smalltalk classes. Because of this symbiosis between CLIC and Smalltalk, the use of CLIC allows taking benefit from modularity and reusability of components without sacrifying performance. Applications in CLIC consume the same amount of memory and CPU as plain Smalltalk applications. Indeed, each clic component is implemented as a single object. Operation calls result into messages looked up by the virtual machine in the classes method dictionary, making processing as fast in CLIC as in Smalltalk.

A second important benefit for the symbiosis is that CLIC components can directly interact with Smalltalk plain objects. Indeed, a Smalltalk object is viewed as a simple component with a unique provided port and no required port. Therefore, CLIC developers can benefit from the full power of Smalltalk class libraries.

Regarding future work, we plan to explore at least two areas of improvements. From the model point of view, we plan to introduce a support for version management. From the tooling point of view, an adapted version of the packaging system is needed. The current implementation of CLIC has been made in Pharo and uses the Monticello packaging systems which is difficult to extend for modified class definitions.

References

E. Bruneton, T. Coupaye, and J. Stefani. Recursive and dynamic software composition with sharing. In *WCOP'02–Proceedings of the 7th ECOOP International Workshop on Component-Oriented Programming*, Malaga, Spain, Jun 2002.

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

Luc Fabresse, Christophe Dony, and Marianne Huchard. Foundations of a Simple and Unified Component-Oriented Language. *Journal of Computer Languages, Systems & Structures*, 34/2-3 (2-3):130–149, 2008. ISSN 1477-8424. doi: 10.1016/j.cl.2007.05.002. URL `http://dx.doi.org/10.1016/j.cl.2007.05.002`.

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. ISBN 1-58113-206-9. doi: http://doi.acm.org/10.1145/337180.337201.

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. ISBN 3-540-61285-8.

Mary Shaw, Robert DeLine, Daniel V. Klein, Theodore L. Ross, David M. Young, and Gregory Zelesnik. Abstractions for Software Architecture and Tools to Support Them. *Software Engineering*, 21(4):314–335, 1995. URL `citeseer.ist.psu.edu/shaw95abstractions.html`.

C. Szyperski. *Component software: beyond object-oriented programming*. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 1998. ISBN 0-201-17888-5.