An Inheritance System for Structural & Behavioral Reuse in Component-based Software Programming

Petr Spacek, Christophe Dony, Chouki Tibermacine, Luc Fabresse

An Inheritance System for Structural & Behavioral Reuse in Component-based Software Programming

Petr Spacek, Christophe Dony, Chouki Tibermacine
LIRMM, CNRS and Montpellier II University
161, rue Ada
34392 Montpellier Cedex 5 France
{spacek,dony,tibermacin}@lirmm.fr

Luc Fabresse
Université Lille Nord de France
Ecole des Mines de Douai
941 rue Charles Bourseul
59508 DOUAI Cedex France
luc.fabresse@mines-douai.fr

Abstract

In the context of Component-based Programming, which addresses the implementation stage of a component-based software engineering development process, this paper describes a specification and an operational integration of an inheritance system into a self-contained new component-based programming language named COMPO. Our proposal completes and extends related works by making it possible to apply inheritance to the full description of components, i.e. both to structural (description of provisions and requirements, of component architecture) and behavioral (full implementations of services) parts in component descriptions. Inheritance in COMPO is designed to be used in conjunction with composition to maximize reuse capabilities and expressive power. COMPO implementation proposes a clear operational solution for inheritance and for achieving and testing substitutions.

Categories and Subject Descriptors D.1.0 [*Programming techniques*]: General—Component based programming technique; D.2.11 [*Software Architectures*]: Languages—Component Based Programming Language

General Terms Component-based, Programming, Language

Keywords Programming, Inheritance, Architectures, Substitutability

1. Introduction

Component-based software is made of off-the-shelf components, connected together into various kinds of architectures. In this domain, several languages, using the component concept as a first class entity in organizing software, have been and are currently designed and prototyped, but some categories have emerged. Architecture Description Languages (ADLs) for models like Fractal [6] or SOFA [15] allow for business-oriented and implementation-independent components and architectures descriptions. Component-Based-Programming Languages (CBPLs) like ACOEL [19], ArchJava [1], CLIC [4], ComponentJ [18], Bichon [23], CompJava [16] or SCL [8, 9] address the implementation stage of a component-based software engineering development process; they allow developers to express full descriptions of executable components.

Inheritance has proved to be one major cornerstone of software reuse, first for the ability it gives developers to organize their ideas on the base of concept classification (a list is a kind of collection, such architecture is a kind of visitor, ...) which is itself one key of human abstraction power and second for the calculus model that makes it possible to not only reuse but adapt software, by executing an inherited code in a new context (the receiver environment).

Many of the above quoted languages somehow propose inheritance mechanisms but they have various limitations (limitation to the architecture description side, limitation to the implementation side which is frequently not achieved with component-based languages, limitation to some part of components descriptions, etc). More generally, the question of the interest of inheritance-based reuse in the component-based software development context is still discussed and has not yet been explicitly nor fully addressed. This papers aims at contributing to that question by proposing a specification and an operational integration of an inheritance system in the context of a component-based programming languages that supports reuse of descriptions of components' structure and behavior. By "structure", we mean provided and required ports together with internal architectural description; and by "behavior", we mean implementations of services that make them executable. Our language, COMPO,

ranges in this category and proposes components as *runtime entities, instances of descriptors*. We introduce an inheritance link between descriptors on which we base an operational system to reuse, i.e. to **inherit**, **extend** and **specialize** descriptions. Descriptors are texts describing components' structure and behavior. Listings 1 and 2 are examples of component descriptors in COMPO.

We address the specific questions of stating what, among port declarations, composition architectures and services definitions, can be inherited, extended or specialized, and how. We notably discuss the rationale and interest of enabling requirements extension or specialization: we will consider various solutions, but will develop an answer to that question which goes in the direction of enabling covariant specialization, because it corresponds to the way human naturally think about concept classification [7] and it promotes expressive power. Our solution will thus propose simple support to help programmers achieve correct substitutions.

An alternative for reuse in the component-based software development context is to use sole composition (see [17, 23] and [22] for a good survey) and message redirection (or forwarding; or delegation[1]); it has the advantage of not introducing any additional mechanism. Our opinion is that composition and inheritance are complementary and that their combination is significantly more efficient especially when structural reuse is to be considered. The above alternative will be further discussed in the related work section, where we show some examples in which sole composition usage leads to code being too complex and reuse being difficult. Inheritance in COMPO is thus designed to be used in conjunction with composition to maximize software reuse capabilities and language expressive power.

The paper is organized as follows. Section 2 proposes an overview of COMPO essential constructions and syntax. Section 3 is a step-by-step (services definitions, requirements declarations, internal architecture description) discussion and presentation of our inheritance system specification illustrated with various examples. Section 4 gives some basic clues on the COMPO's implementation. Before concluding and discussing the future work, we present in Section 5 the related works.

2. A support Component-based programming language

Our proposition for inheritance is demonstrated and validated via its integration into a component-based language named COMPO. Although the language itself is not the subject of this paper, it is needed that we give an overview of its main constructs and syntax. COMPO is an ongoing work that aims at defining in an unified context (1) a component-based architecture description and modeling language that includes all standard component architectures concepts and constructs (eg. component, ports, interfaces, connections, constraints, etc) and (2) a component-based programming language making it possible to write executable applications, in which all the above concepts and constructs are available as first-class entities. This language overall goal is to allow standard applications or architecture verification and transformations applications to be written in the same language, via an integrated meta-level. COMPO is here used for our operational study on inheritance because, as far as we know, there exists no other language in which component architectures and services can be written in the same context and using the same high-level constructs for component-based development. Many component-based architecture are for example translated into Java template code and implemented in a world where concepts such as ports do not have a first-class status.

From a concrete point of view, within a component description (see e.g. listing 1), the `provides`, `requires`, `internally requires` and `architecture` sections are related to components modeling and architecture description. The `service` section is related to services programming; this section can be omitted if the architecture is to be generated into another language.

2.1 Component descriptors.

COMPO applies the descriptor/instance dichotomy where components are instances of **descriptors**. A component descriptor is a text describing the **structure** (ports declarations and architecture description) and the **behavior** (set of services definition) of its instances.

As a first example, Listing 1 shows a version of a `FrontEnd` component descriptor. `FrontEnd` components can be used to build various kind of request servers. This descriptor defines : (1) an external provided port named `default`, providing services `run()` and `isListening()`, (2) an external required port named `backEnd` through which an architect can connect a `FrontEnd` to any other component providing the `handleRequest` service, (3) two internal required ports to achieve internal composition with a `RequestHandler` and with a `TaskScheduler` (note that to satisfy requirements internally or externally is an architectural decision that has several possible solutions, any `FrontEnd` in this example will come equipped with its `RequestReceiver` and `TaskScheduler`). (4) the internal architecture of a `FrontEnd`, (5) the public services of a `FrontEnd`, only one (`isListening()`) being shown. Concepts and syntax used in this example are detailed in the following sub-sections also using the description of an HTTP Server (see listing 2) which uses a `FrontEnd`.

2.2 Ports

In COMPO, a port is a connection point (components are connected through their ports) and a communication point (ser-

[1] Although delegation in prototype-based language is something different, we will use for message redirection the terms "message delegation" and corollary "delegation connector" and "service invocation delegation" with semantics as specified in UML [13]

```
component descriptor FrontEnd {
 provides {
  default : { run(); isListening(); } }
 requires {
  backEnd : { handleRequest(r) } }
 internally requires {
  rR <: RequestReceiver;
  s : TaskScheduler; }
 architecture {
  delegate default to rR@.default;
  connect s to (TaskScheduler new).default;
  connect rR@.scheduler to s@.schedule;
  delegate rR@.handler to backEnd; }
 service isListening() {
       ^ rR isRunning. }
}
```

Listing 1. The `FrontEnd` descriptor

vices invocations are transmitted via ports). A port is defined by a an owner component, a name, a list of service signatures, a visibility (external or internal) and a role (provided or required).

The role of a port is either *required* or *provided* with a standard semantics : a provided (resp. required) port lists, the signature of services offered (resp. required) by a component.

The list of services signatures associated to a provided port can be given :

- as an explicit list (we call such a list an *anonymous interface*) (the `default` port declaration in Listing 1 is an example).
- via a named interface (see Section 3.3),

The list of signature associated to a required port can be given as an explicit list or via a named interface, as above, or via a component descriptor name (e.g. `cd`); in this case, the list is the list of signatures of services associated to `cd`'s default provided port (the `sched` port declaration in Listing 1 is an example).

The visibility of a port is either *internal* or *external* (the default is "external").

- An external port `ep` of a `c` component is visible and can be used by an architect to interconnect `c` into an englobing composition architecture. For example, the external required port `backEnd` of an instance of `FrontEnd` can be connected to the external provided port `default` of an instance of `BackEnd` as shown in listing 2 and in figure 1 (we are developing both textual and graphical component definition interfaces).

  Each component has a default external provided port (named `default`) listing signatures of public services of its owner component, and has a default and unique internal provided port, named `self` (for obvious historical reasons), listing signatures of all services of its owner component and allowing any of its services to invoke another one.

- An internal port `ip` of a `c` component is a support for accessing c's internal composition architecture, `ip` and the component connected to it (that we call an **internal component** of `c`) are invisible from the outside of `c`. In Listing 2, it is shown how the two internal required ports of a `HTTPServer` are connected to its two internal components : a `FrontEnd` and a `BackEnd`, allowing it to invoke their services (e.g. `fE isListening` in service `status`).
- The list of external ports (provided and required) define **the external contract** of a component.

2.3 Connections

A **connection** establishes a dual referencing between two ports, making it possible to determine whether a port is connected or not and, if true, to which other port it is connected. When we sometimes write, in a somehow misleading but admitted way, that two components are connected, it is meant that one port of the former is connected to one port of the later.

- A **regular connection** is between a required and a provided port. We provide two equivalent syntax for connections : `(connect | delegate) <port> to <port>`, or `<port> =:= <port>`, where `<port>` is any expression returning a port. An example of an expression establishing a regular connection is : `connect s to (TaskScheduler new).default;`, (see. Listing 1).
- A **delegation connection** is between two ports having the same role and is used to delegate a service invocation from an external to an internal via the composite (provided to provided), or from an internal to an external via a composite (required to required). An example of a "provided to provided" delegation connection is `delegate default to rR@.default;` in Listing 1).

2.4 Internal architectures

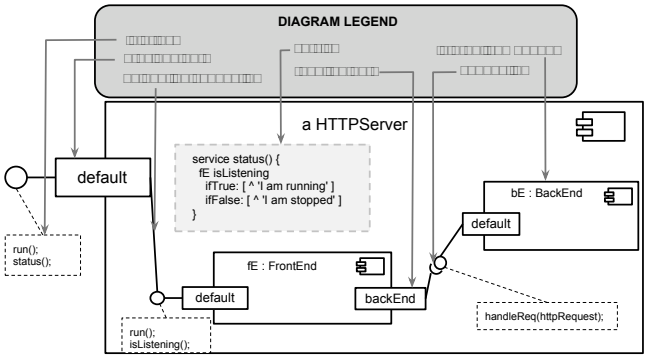

Figure 1. HTTPServer, the diagram shows a logical representation of an instance of the `HTTPServer` descriptor presented in Listing 2, after it has been created and initialized.

An internal architecture of a component is the set of its internal components (components connected to its internal required ports) together with their inter-connections. A component having internal component is as usual called a *composite*. Initialization (i.e. instantiation and connection) of internal components of a composite `c` is performed during c's instantiation if described in the c's descriptor *architecture* section.

For example, the internal ports and architecture sections of descriptor `HTTPServer` define the following internal architecture for its instances.

- The internal required port `fE` is connected to a `FrontEnd`.
- The internal required port `bE` is connected to a `BackEnd`. Note that: `bE <: BackEnd` in the port section is a shortcut to avoid writing the following instruction in the architecture section : `connect bE to (Backend new).default`).
- The internal required port `name` is connected to a `String`. `String` is a predefined component descriptor the role of which is to interface all basic types of the hosting language so that their elements can be used as standard components. Predefined descriptors expose a single provided port named `default` through which procedures or methods of basic types can be called in the form of a service invocation via the COMPO interpretor or virtual machine.
- a delegation connection is created between the provided port `default` and the `default` port of the `frontEnd`.
- a regular connection is created between the required port `backEnd` of the component connected to the `fE` internal required port and the port `default` of the component connected to the `bE` internal required port: `connect fE@.backEnd to bE@.default`.

  This connection is a good example of a situation in which architecture design requires that a refence be made to a component which is not yet created. Here we need to say that we want to connect the port `backEnd` of the component that will be be connected to port `fE`. The `@` operator makes it possible, for any port `p` to reference the component that will later on be connected to `p` and subsequently to specify an architecture before descriptors are instantiated.

2.5 Services & Service invocations

Services implement the behavior of components. A service may have parameters and may return a value.

Components communicate by *service invocations* made through required ports. A service invocation consists of a port name, a selector (the name of the requested service) and optional arguments. `fE islistening` is an example of a service invocation made through the `fE` port in the context

```
component descriptor HTTPServer {
 provides {
  default : { run(); status() } }
 internally requires {
  fE : FrontEnd;
  bE <: BackEnd;
        name <: String; }
 architecture {
        fE =:= (FrontEnd new).default;
  delegate default to fE@.default;
  connect fE@.backEnd to bE@.default; }
 service status() {
  fE isListening
    ifTrue: [ ^ name printString + 'is running' ]
    ifFalse: [ ^ name printString
+ 'is stopped' ]. }
}
```

Listing 2. The `HTTPServer` descriptor.

of the `status()` service of component `HTTPServer`. When a required port receives an invocation it transmits it to the port it is connected to. The receiving port then transfer the invocation to its owner component which is responsible for handling it (either executing a corresponding service or delegating the invocation via a delegation connection, full detail on this can be found in [8]).

3. Rationale and rules to reuse descriptions with inheritance

This section presents the rationale and the operational description of our descriptor-level inheritance embedded in our CBPL. Ports declarations, internal architecture definition and initializations, services definitions; and more generally all pieces of code specified at the descriptor level are subjects to inheritance, extension and redefinition.

For the sake of simplicity, we have limited our proposal to single inheritance relationship.

3.1 Descriptors and basic inheritance

Any descriptor can be defined as a sub-descriptor of an existing descriptor (`D`). In such a case, `D`'s port declarations, internal architecture and service definitions are inherited. Every sub-descriptor has, by default, the `super` internal provided port. The port differs from regular provided ports in one small detail: services implementations demanded by service invocations sent to this port are looked up starting from the super-descriptor of the owning descriptor (not directly from the owning descriptor as it is in the case of regular provided ports).

Listing 3 show the `ControlableFrontEnd` sub-descriptor that extends the `FrontEnd` descriptor (shown in Listing 1) with a new port named `control`. Ports declarations and internal architecture described in the `FrontEnd` descriptor are inherited by the `ControlableFrontEnd` descriptor, this is taken into account when creating an instance of the `ControlableFrontEnd` descriptor whose structure will conform to what is described in the `FrontEnd` descriptor.

In Listing 4 we show an another sub-descriptor named (RestartableFrontEnd) which extends the ControlableFrontEnd descriptor with a new service. Services defined in the FrontEnd and ControlableFrontEnd descriptors are inherited by the RestartableFrontEnd descriptor. This is taken into account when instances of the RestartableFrontEnd descriptor are created and receive services invocations (lookup algorithm).

```
component descriptor ControlableFrontEnd
  extends FrontEnd
{
 provides {
  control : {start(); isRunning(); stop()} }
 architecture {
  delegate control to rR@.control; }
}
```

Listing 3. The ControlableFrontEnd descriptor. Extends the FrontEnd descriptor with a new provided port named control.

```
component descriptor RestartableFrontEnd
  extends ControlableFrontEnd
{
 provides {
  control : { restart(); } }
 service restart() {rR stop. rR start}
 service isListening() {
  super isListening ifTrue: [^0] ifFalse: [^1] }
}
```

Listing 4. The RestartableFrontEnd descriptor. Specializes the control port of ControlableFrontEnd descriptor and service isListening inherited from FrontEnd

3.2 Extension & specialization of services

To be able to inherit, extend and specialize the behavior defined by a component descriptor, a sub-descriptor can introduce new services and its instances can access and reuse services defined by its super-descriptor. This gives us ability to define behavior that's specific to a particular sub-descriptor, i.e. achieve polymorphism of descriptors.

A service can be redefined in a sub-descriptor and can reuse the one it specializes by using the super internal provided port. Sending an invocation to super states that the service implementation should be looked for in the super-descriptor of the descriptor in which the current service in execution has been found.

Listing 4 shows an example of specialization and extension of services, where the descriptor RestartableFrontEnd extends the descriptor ControlableFrontEnd with a new service restart() and specializes service isListening() defined in ControlableFrontEnd, super is used to access ControlableFrontEnd's implementation of the isListening() service.

3.3 Extension & specialization of declarations of provided ports

The basic goal here is to be able to introduce a new provided port declaration and be able to specialize a declaration (i.e. a list of service signatures) of an inherited port. This capability leads to a higher expressive power. For example, sub-descriptors are able to export an internal behavior via newly added ports. Such an export does not break the encapsulation of the internal component, because it exports behavior which has already been public. Another use is definition of more viewpoints for a component, where each provided port represents a viewpoint on the component. Specialization of ports introduces more precise modeling possibilities for software architects. An example of extension and specialization of provided ports is illustrated in Listings 1, 3 and 4, where the ControlableFrontEnd descriptor extends original FrontEnd descriptor with a new port named control. The RestartableFrontEnd descriptor specializes the inherited port control with a new service signature.

The specialization of port roles makes sense only from the required role to the provided role. Indeed, this kind of role specialization can be performed simply by a delegation of a required port to a provided port of a component. Thus, there is no reason for allowing specialization of the roles of ports.

We extend a set of ports simply by introducing a new port in a sub-descriptor. A name of newly added port cannot clash with existing port names. In Listing 3 we extend the original FrontEnd descriptor with a new port named control in order to export control behavior offered by the internal component connected to the rR internal required port.

There are two scenarios how to specialize the list of service signatures of an inherited port: (1) a specialization by adding new service signatures to its list of service signatures (i.e. extending an anonymous inherited interface). The RestartableFrontEnd example in Listing 4 shows the specialization of the control port defined by the ControlableFrontEnd super-descriptor. The specialization is used in order to provide the restart service defined by the descriptor RestartableFrontEnd. (2) a specialization using a named interface. In this case the set of service signatures defined in the named interface has to be a super-set of the set of service signatures used to describe the original port. A specialization of a port named portA looks like: provides { portA : Ispec}. In the super-descriptor, portA was declared by the statement: provides { portA : { ser1(); ser2() }}. The Ispec interface was defined with the statement interface Ispec { ser1(); ser2(); ser3() }. The named interface Ispec defines a set of service signatures, which is a super-set of a set representing the anonymous interface of the original portA port. Therefore it can be used for the specialization.

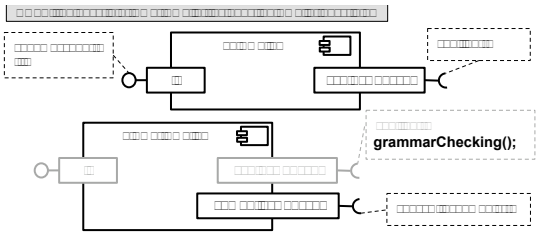

Figure 2. An example of an extension and specialization of required ports. Grayed parts of the figure illustrate inherited parts.

3.4 Extension & specialization of declarations of external required ports

In our inheritance system, we enable adding new declarations of required ports to sub-descriptors and we allow for modification of a declaration of an inherited required port. Extension and specialization of required ports are needed to preserve expressive power of our language. For example, without such a capability, we will not be able to extend the `Emailer` descriptor shown in Figure 2 with a new required port `semanticsChecker` or specialize its required port `syntaxChecker` with a new required service signature `grammarChecking()`. Syntactically these operations do not differ from extension and specialization of provided ports.

```
component descriptor RandomRequestsQueue
 extends RequestsQueue
{
 requires { randomGen : { getNextInt(); }
 ...
}
component descriptor Randomizer {
 provides { generator : { getNextInt(); }
 ...
}

server := QueuedServer new.
assocPair := 'randomGen'->(Randomizer new).
randomQueue := RandomRequestsQueue
  newCompatible: (Array with: assocPair).

server reconnect: 'queue' to: randomQueue.
```

Listing 5. An example of unsatisfied required port problem and its solution using the `reconnect` and `newCompatible` support tools. The `RandomRequestsQueue` descriptor extends the `RequestQueue` descriptor with an additional required port to which an instance of the `RandomGenerator` descriptor should be connected.

Semantically the extension and specialization of required ports introduce a new issue, it **breaks child-parent substitutability**. In OOPLs, substitutability between sub-classes and super-classes is guaranteed. To be more precise, interface compatibility is guaranteed. Behavior compatibility is still not guaranteed, as pointed by [12]. In CBPL with the possibility to extend or specialize super-descriptor's requirements creating a potentially non-substitutable sub-descriptor is possible.

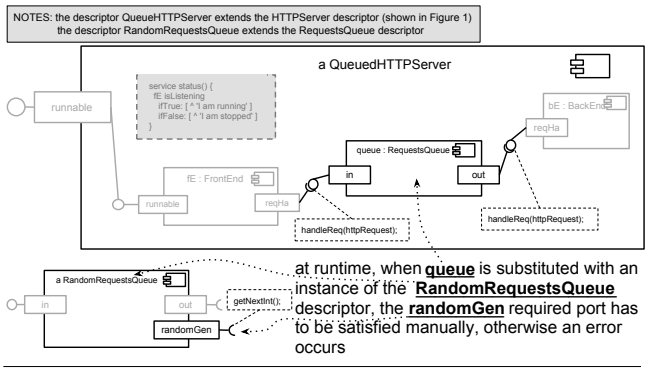

Figure 3. Dynamic substitution with a sub-descriptor having additional required port may lead to unsatisfied requirement in the architecture. Grayed parts of the figure illustrate inherited parts.

Additional required ports may became unsatisfied, when an instance of such a sub-descriptor is used where an instance of the super-descriptor is expected. This may break up the system. In other words, adding new required ports (or specializing a required port in a sub-descriptor) violates the Liskov's substitution principle [11]. These operations change components dependencies and can lead to unspecified behavior. This is illustrated in Figure 3 and Listing 5, where an instance[2] of the `RequestQueue` descriptor is substituted with an instance of the `RandomRequestsQueue` descriptor. The `RandomRequestsQueue` descriptor extends the `RequestQueue` descriptor with an additional required port to which an instance of the `RandomGenerator` descriptor should be connected. At runtime, when an instance of the `RequestsQueue` descriptor is substituted by an instance of the `RandomRequestsQueue` descriptor, the `randomGen` required port may became unsatisfied.

This problem has three possible solutions: (1) forbid extending requirements, but requirements have been made explicit in components and they are considered as important entities to make it possible to introduce new connections. Therefore it is undesirable to limit expressive power of modeling by forbidding extension and specialization of requirements. For example, without possibility to add a required port it is complicated to design the `Emailer` example shown in Figure 2; (2) constrain substitutions - define a rule saying that an original component can be substituted by a new one, only if the new one provides at least the same and requires at most the same as the original one; (3) allow additional requirements and delegate responsibility for additional re-

[2] the instance is the internal component connected to the internal required port `queue` of an instance of the `QueuedServer` descriptor shown in Figure 3 and Listing 6.

quirements satisfaction to the language users, while providing verification support for substitutions.

With COMPO all alternatives are possible, but since the language is oriented toward modeling flexibility, we have experimented with the third alternative. We will thus support covariant specialization if and when needed because it corresponds to the way human naturally think differential description [7]. Our inheritance mechanism does not apply any restrictions to implicitly guarantee substitutability. Substitutions in COMPO are under the developer's control and responsibility.

We are providing three support methods: (1) the *newCompatible* method to return a component compatible with the super-descriptor of the component (the service is automatically created for each sub-descriptor which extends its parent with additional requirements); (2) the *reconnect* method (executed when a substitution is performed) to warn users about unsatisfied additional requirements (by an exception) and (3) the *isCompatibleWith* method to help developers to check the validity of a substitution. For an example of usage see Listing 5.

The first substitutions support comes in case when a sub-descriptor has an additional required port. Then our inheritance system automatically generates a method called `newCompatible:` The method has a unique parameter, an array of pairs *port-component*. The method is able to create an instance, which is substitutable with instances of the super-descriptor. That is all additional requirements are satisfied by connections to components given in the array argument.

The second support is the `reconnect:to:` method to achieve substitutions safely. This method takes two arguments, the first one is the name of an internal required port referencing the component which should be replaced and the second argument is the replacing component. `reconnect:to:` checks for compatibility between the original and the new component descriptor, i.e. checks if the new component provides and requires at least the same as the original one and checks if all requirements will be satisfied after substitution. If all requirements of the new component are satisfied, the replacement is performed, otherwise an exception is thrown. `reconnect:to:` reconnects all ports of the original component to corresponding ports of the new component and connects the new component to the corresponding internal required port of the composite to reference the new component.

For components compatibility checking purposes we provide `isCompatible` method. The `isCompatible` method is able to compare the external contracts of compared components and answer by true if they are compatible and false otherwise. The method is useful for the `reconnect:to:` method, which warns users in the case of unsatisfied requirements.

3.5 Extension & specialization of internal architectures

Internal architecture description and initialization is inherited by a sub-descriptor and it can be extended and specialized. When a large and complicated architecture needs to be reused, the language should support such a feature.

In COMPO, a sub-descriptor may extend a set of internal components by introducing a new internal required port. Usually this action implies extension and specialization of internal connections. These operations are illustrated in Figure 3, where the descriptor `QueuedHTTPServer` extends the descriptor `HTTPServer` with a new internal component named `queue` (described by `RequestsQueue`). And it specializes the inherited connections in order to assemble the `queue` component into the architecture of its super-descriptor. COMPO code of `QueuedHTTPServer` is given in Listing 6.

Specialization of an inherited internal component in a sub-descriptor can be achieved by modifying the interface description of the internal required port associated with the internal component. The descriptor of `PriorityQueuedServer` in Listing 6 specializes an inherited internal component `queue` by describing it with the `PriorityRequestsQueue` descriptor .

The specialization of an inherited internal connection can be achieved by the combination of statements having the following syntax: `disconnect <port-name> from <port-name>` and `connect <port-name> to <port-name>` (statements were explained in Section 2.5).

```
component descriptor QueuedHTTPServer
 extends HTTPServer
{
 internally requires {
  queue <: RequestsQueue }
 architecture {
  disconnect fE@.backEnd from bE@.default;
  connect fE@.backEnd to queue@.in;
  connect queue@.out to bE@.default; }
}
component descriptor PriorityQueuedServer
 extends QueuedHTTPServer
{
 internally requires {
  queue <: PriorityRequestsQueue }
}
```

Listing 6. Specialization and extension of an internal architecture.

4. Implementation

The current implementation of COMPO is used as a laboratory for exploring new ideas and is built in [3] as an extension of the SCL [8, 9] implementation in the context of a global effort towards the development of efficient dynamic languages.. We have chosen Smalltalk because we are a part of the effort to bring component concept into the Pharo environment. Descriptors and sub-descriptors are implemented as subclasses of the `CompoComponent` class. The

`CompoComponent` class contains the mechanism to store necessary inforation about ports and associated interfaces, internal components and connections. The inheritance mechanism uses Smalltalk's meta-model facilities to implement extension and specialization operations for both the structure and behavior of descriptors.

Readers can download a Pharo image of COMPO implementation here: http://www.lirmm.fr/∼spacek/compo/

5. Related works

	CBPLs					ADLs	
criterium/model	ACOEL	ArchJava	CLIC	CompJava	COMPO	Fractal	SOFA
Structure inheritance	yes	yes	yes	yes	yes	yes	yes
Behavior inheritance	yes	yes	yes	no	yes	no	no
Extensions							
Provided ports	yes	yes	no	yes	yes	yes	yes
Required ports	yes	yes	yes	no	yes	yes	yes
Internal components	yes	yes	yes	no	yes	yes	yes
Connections	yes	yes	yes	no	yes	yes	yes
Specialization							
Provided ports	yes	no	yes	yes	yes	yes	yes
Required ports	yes	no	yes	no	yes	yes	no
Internal components	no	no	yes	no	yes	yes	yes
Connections	no	no	yes	no	yes	yes	yes
Substitution:							
restrictive	yes	yes	yes	yes	yes	yes	yes
with addit.reqs.	no	no	yes	no	yes	no	no

Table 1. Comparative table of inheritance in selected CBPLs and in Fractal and SOFA models

In this section we give an overview of how inheritance is used in existing Component-based Programming Languages (CBPLs) and ADLs, and compare this with our proposal. We also compare our proposal based on a combination of inheritance and composition with inheritance-free proposal in which all reuse schemes are achieved using sole composition.

It is important to note, that none of these languages we have studied propose a complete specification of inheritance which concerns all main reuse aspects. Our knowledge about the behavior of their inheritance mechanism often had to be extracted from experiments conducted using these languages. The list of related works is not exhaustive; especially ADLs that do not support any form of descriptors inheritance are not included.

5.1 Inheritance in related CBPLs

Here, we compare how related CBPLs integrate inheritance aspects such as: the structure inheritance , the behavior inheritance and abilities to extend and specialize particular definitions in a component descriptor (i.e. ports, internal components and connections.) As related CBPLs we consider ACOEL [19], ArchJava [1], CLIC [4] and CompJava [16], because these languages combine implementation and architecture specification.

Structure inheritance is partially supported in all other languages. We say partially, because CompJava do not allow the reuse of internal components and connections specification. Ports declarations can be reused via component type definition. Except that they use a different terminology, the languages define component type as a set of port names including interface references and roles specification. And then a component type can be defined as an extension of an existing component type

Behavior inheritance is fully supported only in CLIC and ArchJava languages. ACOEL model supports implementation inheritance by the *extend* statement, but a child cannot access any of the internals (implementation classes, methods) of a parent, except via the input ports of the parent, i.e. `this.<portname>.<servicename>` (composition-like approach). The advantage of this black-box approach is that it preserve encapsulation of parent components. We support white-box approach to be able to specialize services implementations which are not provided by a parent.

Ports specialization is not supported in ArchJava, because adding new provided methods to an existing port might cause ambiguities if these provided methods were required by a connected component, and provided by a different component. There would then be two components providing the same required method, breaking ArchJava's connection rules. Adding required methods to an existing port would make the component class non-substitutable for the component superclass, because connections made to the superclass might not provide the subclass's required methods. Required methods in a new port are also problematic, because the new port might not be connected at all.

An interesting solution comes with ACOEL model, which uses ports parametrized by mixins [5]. During instantiation a mixin can be passed as an argument of a component constructor and a parametrized port is then "decorated" with new behavior. In COMPO, ports are just descriptions of communication and connection points. Therefore, they should not carry any implementation of behavior or its specialization.

Ports extension is well supported. CLIC model does not support additional provided port, because this model allows components to have only one provided port. The idea of a single provided port is based on the observation that developers do not know beforehand, which services will be specified by each required port of a client component. Therefore it is hard to split component functionality over multiple ports. We see this as a unnecessary limitation of modeling power.

On the other hand, in the CompJava, a component type may extend another component type and it inherits all ports. It may extend the interface of inherited provided ports or may add provided ports. Extension of required ports is not allowed due to the substitutability policy of the CompJava model.

Architecture extension and specialization. ACOEL and ArchJava treat internal components as regular instance variables of classes and therefore there is no way to specialize inherited internal components. CompJava supports inheritance

of component types only. Component types do not involve internal components and connections declarations, therefore architecture cannot be reused.

Substitutions. ACOEL, ArchJava and CompJava define sub-type relation as defined by Liskov [11]. In general, a component type is a sub-type of another one if it provides at least the same and requires at most the same. To ensure ACOEL use a type system checking. CompJava and ArchJava forbid additional requirements (in inherited types) and then they restrict substitutability by the sub-type relation.

In Table 1, we make a summary of this comparison.

5.2 Inheritance in related ADLs

Structure reuse is a primary inheritance property supported by ADLs [14], which operate mainly at design stage in the development process. They make it possible to design software using components and then convert them into a component framework written in OOPLs or other programming languages (but not CBPLs). ADLs generally do not support behavior inheritance, because behavior of components is specified either in programming languages they are implemented in, or using some formalisms where there is no inheritance.

For example Fractal and its Fractal ADL allow to extend one component definition with another one, then a sub-definition can add or override elements. It also extends component type definition with contingency (optional or mandatory) and cardinality for each port. It is possible to add connections but impossible to specialize an inherited connection.

SOFA Component Definition Language (CDL) uses the *frame* term for component types. One frame can inherit from another frame and then port declarations are reused. To compose several frames, SOFA introduce *architecture* construct, where an architecture implements a frame and may inherits from an another architecture. In this way, internal components and connections are reused. Ports are specialized using interface redefinition i.e. by the following statement `frame ComponentName inherits InheritedCompName changes InterfaceInstance1:: OriginalInterfaceType1 => NewInterfaceType1`. Specialization of inherited connections is supported by the statement: `newTie1, newTie2 replacing originalTie subsume subcompInstName:intInstName to intInstName exempt: subcompInstName:intInstName`.

Other ADLs do not specify inheritance between descriptors, they usually use inheritance uniquely for creation of sub-interfaces. In UML, the component entity inherits from the structured class entity and therefore they can participate in generalization relationship in the same way as classes do.

5.3 Composition as a reuse mechanism

In ComponentJ [18] the authors state that all reuse schemes are achieved using sole composition. By nature, the composition concept allows only black-box reuse. Therefore it

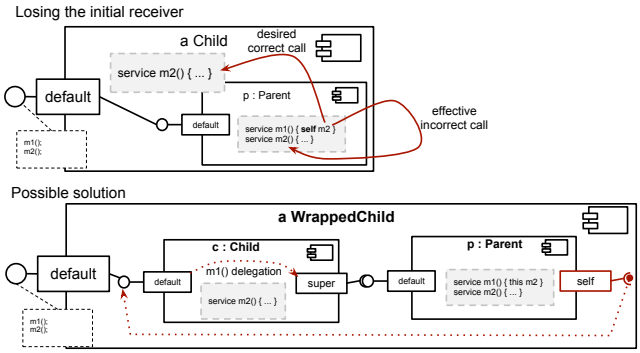

Figure 4. The "lost of initial receiver" problem, when composition and invocations forwarding are used. The context pseudo variable `self` does not refer to the original receiver of service invocation when delegations are used. The `m2()` service of descriptor `Child` is never called

cannot be used when an internal architecture of a descriptor needs to be reused. For example the reuse of a descriptor's internal architecture shown in Figure 3 cannot be performed with composition.

Achieving behavioral inheritance by using composition and message forwarding raises various issues including the well known "lost of initial receiver" problem [10]. The context reference `self` (or `this` in some languages) does not refer to the original receiver of service invocation when forwarding is used. The problem and its solution are illustrated in Figure 4, where the `m2()` service of descriptor `Child` is never called.

ComponentJ proposal describes an operational solution to that problem. The solution is based on a connexion of the "self" port of the composed-component to the default provided port of the composite and on the service invocations forwarding from the composite to the composed-component (see the dashed arrow in the bottom of Figure 4).

The solution is operational but we argue that inheritance is preferred because it: (1) enables abstract description and conceptual description; (2) does not increases the complexity of the system and therefore preserve code maintainability of large systems, where hierarchical concept modeling is used; (3) copy the structure of an external contract of a super-descriptor (i.e. declaration of provided and required ports) automatically to sub-descriptors.

6. Conclusions

In this paper, we have proposed an original descriptor-based inheritance system for a component-based programming language taking into account in the same context the architecture modeling and coding aspects of components. We have motivated and described concrete solutions for creating new component descriptors by extending and specializing existing ones and for extending or specializing all

key primary aspects of component modeling and programming: ports declarations, composition (or connection) architectures, services declarations and definitions. These solutions are implemented and can be tested with our prototype (http://www.lirmm.fr/∼spacek/compo/) implemented in Pharo Smalltalk [3].

Among the various possible specialization options for inheritance, we have applied in our context the covariant-oriented one, by allowing to add requirements on a sub-descriptor, that promotes expressive power and reuse. We have implemented supports to help programmer achieving substitutions. Other well-known approachs can of course be considered. Besides, we have not yet discussed the multiple-inheritance option. Inheritance in component-based development is useful for both "development-for-reuse" and "development-by-reuse". It helps, in combination with other composition techniques, in effortless production of new off-the-shelf components or new applications embedding complex components architectures.

There are various prospective for this work. The first one is to obviously to gain more experience on using inheritance in this context and to consider its use in designing Component Design Patterns [2]. Beside, our inheritance extension is currently used to build a reflective version of COMPO, the primary goal of which is, through a first-class representation of descriptors, components, ports and architecture, to write model-driven verifications or transformations of COMPO component-based applications in COMPO itself. We are applying it to constraint-based architecture verification, extending our earlier works [20, 21] towards constraint-components hierarchies.

References

[1] J. Aldrich, C. Chambers, and D. Notkin. Archjava: connecting software architecture to implementation. In *procs. of ICSE*, New York, NY, USA, 2002. ACM.

[2] K. Arnout. From patterns to components. ETH Zürich, 2004.

[3] A. Black, S. Ducasse, O. Nierstrasz, D. Pollet, D. Cassou, and M. Denker. *Pharo by Example*. Square Bracket Associates, 2009.

[4] N. Bouraqadi and L. Fabresse. Clic: a component model symbiotic with smalltalk. In *procs. of IWST*, New York, NY, USA, 2009. ACM.

[5] G. Bracha and W. Cook. Mixin-based inheritance. In *Proceedings of OOPSLA and ECOOP*. ACM Press, 1990.

[6] E. Bruneton, T. Coupaye, M. Leclercq, V. Quéma, and J.-B. Stefani. The fractal component model and its support in java: Experiences with auto-adaptive and reconfigurable systems. *Softw. Pract. Exper.*, September 2006.

[7] R. Ducournau. "real world" as an argument for covariant specialization in programming and modeling. In *Advances in Object-Oriented Information Systems*, Lecture Notes in Computer Science. Springer Berlin / Heidelberg, 2002.

[8] L. Fabresse, C. Dony, and M. Huchard. Foundations of a simple and unified component-oriented language. *Comput. Lang. Syst. Struct.*, July 2008.

[9] L. Fabresse, N. Bouraqadi, C. Dony, and M. Huchard. Filling the Gap between Design and Implementation with Components. *International Journal of Computer Languages, Systems and Structures*, Jan. 2012.

[10] H. Lieberman. Using Prototypical Objects to Implement Shared Behavior in Object Oriented Systems. In *procs. of OOPSLA*, Portland, Oregon, USA, Nov. 1986. Published as ACM SIGPLAN Notices 21(11).

[11] B. Liskov and S. Zilles. Programming with abstract data types. In *Proceedings of the ACM SIGPLAN symposium on Very high level languages*, New York, NY, USA, 1974. ACM.

[12] R. C. Martin. *Agile Software Development, Principles, Patterns, and Practices*. Prentice-Hall, Inc, 2002.

[13] OMG. UML 2.4.1 superstructure specification; document formal/2011-08-06. Technical report, OMG, August 2011.

[14] T. Oplustil. Inheritance in architecture description language. In *procs. of WDS*, Prague, Czech Republic, June 2003. Matfyzpress.

[15] F. Plásil, D. Bálek, and R. Janecek. Sofa/dcup: Architecture for component trading and dynamic updating. In *procs. of CDS*, Washington, DC, USA, 1998. IEEE Computer Society.

[16] H. A. Schmid and M. Pfeifer. Engineering a component language: Compjava. In *Software and Data Technologies*, Communications in Computer and Information Science. Springer Berlin Heidelberg, 2008.

[17] J. a. C. Seco and L. Caires. A basic model of typed components. In *procs. of ECOOP*, London, UK, 2000. Springer-Verlag.

[18] J. C. Seco, R. Silva, and M. Piriquito. Componentj: A component-based programming language with dynamic reconfiguration. *Computer Science and Information Systems*, Dec. 2008.

[19] V. C. Sreedhar. Mixin'up components. In *procs. of ICSE*, New York, NY, USA, 2002. ACM.

[20] C. Tibermacine, R. Fleurquin, and S. Sadou. A family of languages for architecture constraint specification. *In the Journal of Systems and Software (JSS), Elsevier*, 2010.

[21] C. Tibermacine, S. Sadou, C. Dony, and L. Fabresse. Component-based specification of software architecture constraints. In *procs. of CBSE*, Boulder, Colorado, USA, June 2011. ACM Press.

[22] W. Weck and C. Szyperski. Do we need inheritance? In *procs. of ECOOP*, 1996.

[23] L. Xu and Y. Ren. Bichon: A new component-oriented programming language. *Software Engineering, World Congress on*, 2010.