Bridging the Gap between Component-based Design and Implementation with a Reflective Programming Language
Petr Spacek, Christophe Dony, Chouki Tibermacine, Luc Fabresse

To cite this version:

Petr Spacek, Christophe Dony, Chouki Tibermacine, Luc Fabresse. Bridging the Gap between Component-based Design and Implementation with a Reflective Programming Language. RR-13028, 2013. lirmm-00862477

HAL Id: lirmm-00862477
https://hal-lirmm.ccsd.cnrs.fr/lirmm-00862477
Submitted on 16 Sep 2013

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L’archive ouverte pluridisciplinaire HAL, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d’enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.
Bridging the Gap between Component-based Design and Implementation with a Reflective Programming Language

Abstract

Component-based Software Engineering studies the design, development and maintenance of software constructed upon sets of connected components. Existing component-based models are frequently transformed into non-component-based programs, most of the time object-oriented, for runtime execution and then many component-related concepts, e.g. explicit architecture, vanish at the implementation stage. The main reason why is that with objects the component-related concepts are treated implicitly and therefore the original intentions and qualities of the component-based design are hidden. This paper presents a reflective component-based programming and modeling language, which proposes the following original contributions: 1) Components are seen as objects in which requirements, architecture descriptions, connection points, etc. are explicit. This core idea aids in bridging the gap between component-based modeling and programming; 2) It revisits standard solutions for reification in the context of components when using the component-oriented reification to build up an executable meta-model designed on the idea of "everything is a component", allowing intercession on component descriptors and their instances; 3) It integrates reflection capabilities, making it possible to develop standard component-based application, but also to perform advanced architecture checking, code refactoring or model transformations using the same language.

Categories and Subject Descriptors D.1 [Programming techniques]; D.2.11 [Software Architectures]: Languages; D.3 [Programming languages]

1. Introduction

Research works on component-based software engineering (CBSE) have brought many advances on how to achieve complex software development by reusing and assembling components. The current trend is to explicitly express architectures of software solutions, to reason about them, to verify them and to transform them. However it appears that component-orientation has been more studied at design stage, with modeling languages and ADLs [10, 16, 22] rather than implementation stage. As stated in [10] ‘most component models use standard programming languages ... for the implementation stage’; and most of today’s solutions [13] use object-oriented languages. Such a choice has many practical advantages related to the availability and maturity of object-oriented programming languages, environments, tools and practices. But this also has the important global drawback that component-related concepts (such as component descriptor, ports, component, internal component, internal architecture, etc) vanish at the implementation stage. Seeing the modeling and programming stages as two isolated activities leads to complicated relationships between the artifacts that are produced. There is in addition a great risk of inconsistencies between artifacts, and often the result is that the models are discarded and the program becomes the only artifact for subsequent development. Also, modeling is hampered by poor tool support compared with programming tools. Such a lack of a conceptual continuum between various development stages is the source of various issues.

- It makes debugging or reverse-engineering (e.g. from implementations to models) complex.
• It can entail some loss of information or some inconsistencies when implementing a model, such as the violation of the communication integrity [13].

• Different languages have to be learned and mastered to write an application e.g. an ADL for the architecture, a programming language for the implementation (model transformations only generate skeleton implementations), a language for expressing architecture constraints (such as OCL) and possibly a language for model transformations [27].

• Some pieces of code may have to be written twice, for example, in the absence of an automatic transformation of model-level constraints [31], the same constraint expression to check \(^1\), for example, that a given port of a component is correctly connected has to be written differently if it has to be checked at the model level (using elements of the component description language meta-model - e.g. \texttt{component.port.isConnected()}\(^2\)) or at the implementation level using elements of the implementation language meta-model, (if they are reified).

• Dynamic (runtime) constraints checking is only possible if the implementation language has an executable meta-model that allows for introspection. For example if the implementation language is Java, constraints-checking expressions can be written using the \texttt{Reflect} package.

• Runtime model transformations, to dynamically adapt models or programs to a changing context, are only possible if the implementation language has an executable meta-model that allows for intercession. Furthermore, after such a transformation, a reverse-engineering is needed to update the model.

This above list of issues first suggests to study the combination of today’s advances in CBSE with a development framework including languages that all support CBSE, then offering a conceptual continuum between designs and implementations, similar to the continuum that exists between object-oriented design and implementation. The study of component-based development languages [1, 13, 28, 30] is a step in such a direction.

This list secondly suggests that this principle could also encompass the activity of writing all kind of meta-programs. This globally means to allow software engineers to achieve, using the same language defined by a unique component-based meta-level \(M\), not only applications (architectures and code) but also all those meta-programs, e.g. constraint-checking or model transformation or program transformation programs, that use or manipulate \(M\) constitutive elements and their instances, either statically or at runtime. One step further, it appears that a component-level reflective development language is a possible original solution to such a requirement. By “component-level” we mean a reflective language based on a meta-model describing component related concepts.

A reflective solution provide means to drastically reduce the issues by having the same description of architecture at design and run-time. This paper present such a solution in a form of a reflective component-based programming language named \texttt{COMPO}, that apply existing solutions in the component-based context, and allows for writing architectures, code, program verification and transformation in the same language. \texttt{COMPO} achieves a reification of elements of a new component-oriented meta-model, structurally inspired by [8], designed on the idea of “everything is a component”, to build up an executable meta-model, allowing introspection and intercession on programs elements concepts and their instances. It can be used at all stages of components development to manipulate standard and “meta”-components as first-class entities. It simply opens the possibility that architectures, implementations and transformation can all be written at the component level and possibly (but not manda-
torily) using a unique language (like \texttt{COMPO}).

The paper is organized as follows. Section 2 presents \texttt{COMPO}'s standard syntax, constructs and use, necessary to the understanding of the later examples; Section 3 describes \texttt{COMPO}'s reflective meta-model and some primary examples of its interest; Section 4 describes \texttt{COMPO} implementation; Section 5 presents two examples of use : the reflective integration of constraints components as defined in [32] and an example of model transformation. Comparison with related works is presented in Section 6 and we conclude in Section 7 by discussing future work.

2. \texttt{COMPO}'s basics

Before discussing and describing the reflective version of our language, it is needed that we give an overview of its basic constructs and syntax. \texttt{COMPO} component’s

\footnote{It can be expected that a model transformation ensures that one constraint verified at the model level be by definition also verified at the implementation level but this can only be generally true if the meta-model also describes the instructions of the programming language.}

![Diagram of an HTTPServer component instance](image)
model is described by the MOF meta-model in Figure 2\(^2\). This component model is based on a descriptor/instance dichotomy where components are instances of descriptors. At a glance, the Listing 1 shows a definition of a descriptor named HTTPServer modeling very simple HTTP servers. It defines a default provided port through which it provides the services run and status. It states that a server is composed of two internal components, an instance of FrontEnd accessible via the internal required port fE, and an instance of BackEnd accessible via the internal required port bE. These internal components are connected together so that the front-end can invoke services of the back-end. The HTTPServer descriptor explicitly defines the implementation of the status service. The provided service run is implemented by a delegation connection to the provided port default of the front-end. Figure 1 shows a diagram that represents a component, instance of the HTTPServer descriptor.

```
Descriptor HTTPServer {
  provides {
    default : { run(); status() }
  }
  internally requires {
    fE : FrontEnd;
    bE : BackEnd;
  }
  architecture {
    connect fE to default@(FrontEnd.new());
    connect bE to default@(BackEnd.new());
    delegate default@self to default@fE;
    connect backEnd@fE to default@bE;
  }
  service status() {
    if(fE.isListening()) {
      [return name/printString()
        +'is running']
    } else {
      [return name/printString()
        +'is stopped'];
    }
  }
}

Listing 1. The HTTPServer descriptor.
```

Let’s look at each point more precisely. A descriptor defines the structure and behavior of its instances. The behavior is given a set of service definitions, for example a part of an HTTPServer’s behavior is defined with the status service. The structure is given by descriptions of ports and connections. Descriptions of external (resp. internal) ports define an external contract (resp. an internal contract). For example the external contract of HTTPServer instances is defined by the declaration of the provided port default and its internal contract is defined by the declaration of the fE and bE internal required ports.

A component may be composed of (internal) components (e.g. a HTTPServer is composed of an instance of FrontEnd connected to an instance of BackEnd) and it is then called a composite. A composite is connected to its internal components via its internal required ports. The services of a composite can then invoke the services of its internal components through such ports. The system composed of internal components and their connections is called the internal architecture of a composite. An example is given in the architecture section in Listing 1.

Ports realize port descriptions (similarly to slots realizing classes’ attributes in UML [22]). A port has a role (provided or required), a visibility (external or internal), a name and an interface. An interface is a set of service signatures which could be given in three forms: (i) as an explicit list (we call such a list an anonymous interface), for example the default port declaration in Listing 1; or (ii) via a named interface, e.g. printer : IPrinting where the interface IPrinting was created with the statement: interface IPrinting {print(text); ....}; or (iii) via a 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 fE port declaration in Listing 1 is an example). External ports are visible from the outside environment and are used for communicating with neighboring components in the environment. Internal ports of a component are used for communication with internal components (the owner). Internal ports and the internal architecture of the owner are not accessible from the outside environment.

Ports are connection and communication points. Components are connected through their ports (to say that components are connected is an admitted shortcut to say that one port of the former is connected to one port of the later). Communications happen through ports. A service invocation is made via a required port and transmitted to the provided port, the required port is connected to. Required ports are communication points through which a component invokes services it requires. fE.isListening() is an example of a service invocation expression in the code of the status() service defined in the HTTPServer descriptor, made through the fE required port.

Connection are either regular or delegation 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. It is a 1:1 relationship. We have introduced collection ports as a support for 1:N relationships. A required port can be declared as a collection port (syntax is <portName> []) meaning that the port can be connected to one or more provided ports accessible through an index. Provided ports can also be collection ports which can be connected to one or more required ports.

\(^2\)Only important MOF attributes and operations of COMPO concepts, useful for explaining our contribution in this paper, are shown in this figure, and presented in the text.
The syntax for connections is: (connect | delegate) <port> to <port>, where <port> is any expression returning a port. An example of an expression establishing a regular connection is: connect backEnd@fE to default@bE; (see. Listing 1). The expression backEnd@fE should be read: "the port backEnd of the component that will be connected to fE port after an instance of HTTPServer descriptor will be created", i.e. the @ operator makes it possible to reference ports of a component which is not yet created.

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 port (provided to provided), or from an internal to an external port (required to required). An example of a "provided to provided" delegation connection is delegate default@self to default@fE; in Listing 1.

Finally COMPO has an inheritance system [30], see the "inherits" relation of Descriptor in Figure 2. A descriptor can be defined as a sub-descriptor of an existing one using the extends statement. Sub-descriptors inherit descriptions given by their super-descriptor and can extend or specialize them. Descriptors and inheritance are two important characteristics for our integration of reflection in the language.

3. A meta-model for reflection and the reflective implementation of its elements

This section presents an adaptation and extension of the meta-model presented in Section 2 to allow for structural reflection3, i.e. "to provide a complete reification of both a program currently executed as well as a complete reification of its abstract data types" [11]. Reification can be seen as a process that makes meta-model elements accessible (read access in the case of introspection or read/write in the case of intercession) at the model level (or programming level). The MOF meta-model presented in Figure 3 describes how its elements, representing the main component-level concepts, are organized to be further reified as first-class entities accessible in COMPO’s programs. Reification supposes to solve various potential infinite regressions; in our component context, the key issues are related to descriptors, ports and connectors (or connections).

Our modeling scheme to represent descriptors as components is directly inspired from [8] and conforms to the MOF solution for reflection where “Reflection introduces Object as a supertype of Element in order to be able to have a Type that represents both elements and data values. Object represents ‘any’ value and is the equivalent of java.lang.Object in Java.” [21]. Component in figure 3 is our root classifier, that conforms to MOF::Reflection::Object. Descriptor is our basic meta-classifier, that conforms to UML::Classes::Kernel::Classifier. To keep our contextual component-level terminology, the modeling scheme is all elements in Figure 3 are descriptors. Descriptor is the descriptor of descriptors, all descriptors are instances of it. All descriptors inherit from Component (except Component itself which is the root of the inheritance tree). All descriptors are components. Descriptor is instance of itself, it is its own descriptor. This solves at the model level the infinite regression on descriptions, the corresponding solution at implementation level is to create by hand a bootstrap first version of Descriptor, the implementation of Descriptor.

Ports being true components is important for model checking and transformations and also to allow for defining new kind of ports introducing new communications protocols. It however induces two potential infinite regressions. The former is related to the definition: “a port is a component having ports”. To solve the recursive nature of that definition we restrict the language capabilities by altering the definition in the following way: “a port is a component having primitive ports”. A primitive port is a rock-bottom entity that cannot be created by users and cannot be used as a first-class

Footnotes:
1. Our global solution makes it possible to define new kind of ports (see, the example later on in this section) in which service invocation can be modified. This is a very limited kind of behavioral reflection. More globally, considering the requirements advocated in this paper, structural reflection only is advocated.
2. this is an excerpt of the complete meta-model that only presents its central “interesting” parts)
entity. All ports of any normal port are automatically created as primitive ports. The latter is related to the fact that if ports are components, a component and one of its ports, should be connected via ports. To solve this, the attachment of a port to its owning component has to be primitive and in conjunction a language special construct is needed to provide access to a port seen as a component. COMPO’s such construct is presented in section 3.2.

Similar issues would apply with first-class connectors in the case where component are directly connected via connectors. Having a solution where components are connected via theirs ports, we can consider connections between ports as primitive entities (references), and do not need to reify connections. This entails no limitation regarding the capability to experiment with various kind of connections [18] because our model makes it possible to define new kind of ports (see section 3.2) and because of the capability it offers to put an adapter component in between any components.

The following sub-sections describe the COMPO’s reflective implementation of the main meta-model elements, describe the associated language constructs and give some primary examples of their use. Each element of our meta-model is implemented as a COMPO descriptor. The inheritance relations in the meta-model are almost directly implemented in COMPO using its descriptor-level inheritance system and its ability to create sub-descriptors of descriptors [30].

### 3.1 First-class descriptors and components

The Component descriptor, root of the descriptors inheritance tree, defines the basic structure and behavior shared by all components. Its reflective definition in COMPO (cf. Listing 2) shows that all have an external provided port named default described by the universal interface * and two internal provided ports named self and super allowing a component to invoke its own services (defined in its own descriptor or the services its descriptor has overridden). Component also defines four basic services:

- `getPorts()` and `getPortNamed(name)` return all (external and internal) ports (resp. a particular port) owned by the receiver.
- `getDescriptor()` returns the receiver’s descriptor.
- `getOwner()` returns the owning component of the receiver or null if the receiver is not an internal component.

Listing 3 shows COMPO definition code of the Descriptor descriptor. The Descriptor descriptor extends Component. Its definition states (cf. Listing 3) that all descriptors have, in addition to what is defined in Component, four internal required ports:

![Figure 3. An Excerpt of the meta-model of Compo showing the integration of reflection](image-url)
• name,
• ports, a descriptor has a collection of port’s descriptions (instances of PortDescription) according to which ports of its instances will be created,
• architecture states that a descriptor has a description of its instances internal architecture in the form of a collection of connection’s descriptions (instances of ConnectionDescription) according to which its instances will be initialized,
• services to store the collection of services of its instances.

Descriptor defines services for instance creation, new to create any anonymous component and newNamed(name, super-desc) to create new descriptors, for introspection (various read-accessors such as getDescribedPorts()) and for intercession (such as addService(service)). These services, together with those inherited from Component, set the basis for creating more complex reflective operations.

Descriptor Descriptor extends Component
{
    internally requires {
        name : IString;
        ports[] : {
            getName();
            getRole();
            ...
        };
        architecture[] : {
            getSrc();
            getDest();
            ...
        };
        services[] : { execute();...};
    }
    service new() {...}
    service newNamed(name, super-desc) {...}
    service getDescribedPorts() {...}
    service addService(service) {...}
    ...
}

Listing 3. The Descriptor descriptor.

An introspection example The following code snippet shows a basic use of introspection. The expression returns the descriptions of ports default, self and super, which are defined by the descriptor Component, see Listing 2.

Component.getPortNamed('default').getDescribedPorts();

An intercession example The following code snippet shows the descriptor (named ServiceMover) of a refactor-

Descriptor ServiceMover {
    requires {
        srcDesc : IDescriptor;
        destDesc : IDescriptor
    }
    service move(serviceName) {
        srv := srcDesc.getService(serviceName);
        destDesc.addService(srv);
        srcDesc.removeService(serviceName);
    }
}

An example of defining a meta-descriptor Descriptor is a meta-descriptor. New meta-descriptor can be defined by extending it. As an example, consider the following issue. Having an inheritance system, it is possible for a sub-descriptor SD to define new required ports, thus adding requirements to the contract defined by its super-descriptor D. In such a case, substitution of an instance of D by an instance of SD needs specific checking (child-parent incompatibility problem [30] of inheritance systems in CBSE). It may be wanted to define some descriptors that do not allow their sub-descriptors to add new requirements. Such a semantics is achieved by the DescriptorForSafeSubstitution definition shown in the following code snippet. The meta-descriptor extends the descriptor Descriptor and specializes its service addPortDescription, which implements the capability to add a port description. The service is redefined in a way that it signals an exception each time it is tried to add a description of an external required port.

Descriptor DescriptorForSafeSubstitution extends Descriptor
{
    service addPortDescription(portDesc) {
        req ext |
        req := portDesc.isRequired();
        ext := portDesc.isExternal();
        if (req & ext)
            [self.error('no new reqs. allowed')]
        else {
            super.addPortDescription(portDesc) }
    }
    ...
}

An instance (a new descriptor) of the DescriptorForSafeSubstitution meta-descriptor named TestDescriptor extending descriptor Component could then be created by the following expressions:

• Run-time creation
    DescriptorForSafeSubstitution.
    .newNamed('TestDescriptor', Component);
• Static creation

DescriptorForSafeSubstitution TestDescriptor
extends Component
{
  ...
}

To conclude this part on components and descriptors, let us note why PortDescription and ConnectionDescription definitions are not shown. They simply declare a single provided port through which they offer getter and setter services for accessing the descriptor level descriptions of ports and connections. Such descriptions are useful to achieve static or dynamic architecture checking or transformation. In the case of a runtime transformation implementation should ensure that these descriptor level descriptions and descriptor instances internal representation are causally connected. When the description changes, all instances should automatically be updated.

---

Descriptor Port extends Component

  requires {
    owner : IComponent
    connectedPorts[] : IPort
  }

  internally requires {
    name : IString;
    interface : IInterface
  }

  service getName(){...}
  service getInterface(){...}
  service invoke(service){...}
  service isConnected(){...}
  service connectTo(port){...}
  service disconnect(index){...}
}

Listing 4. The Port descriptor.

3.2 First-class ports

Generally the “port” concept is a higher-level abstraction of the reference concept known from OOP. Using the OOP terminology we can say that required ports represent the fact that one port references another one and provided ports represent the fact that a component is referenced. In the component context with explicit ports, having first-class ports opens, in an explicit and simple way, the door to application scenarios similar to the ones in the object-oriented context, where first-class references are introduced [2].

In the previous, we have explained how our solution for ports reification is based on two main concepts, Port and PrimitivePort. Primitive ports are not reified; they are implemented at COMPO’s virtual machine level. The listing 4 shows the COMPO’s definition of the Port descriptor that implements the Port concept. Each port has an owner, any port is owned by a component, and a connectedPorts to which it can be connected, a name and an interface. Port defines services for ports introspection (e.g. getName(), ...), ports intercession (e.g. connectTo(port), invoke(service), ...).

As explained above, the attachment of a port to its owning component has to be primitive to avoid an infinite number of connections and to allow for efficient service invocation. Services invocation are made via ports, for example the expression printingPort.print('hello'), where printingPort is a port of a component c, will invoke the service print of the component connected to c via printingPort. To use printingPort as a component, to send it a service invocation for example, requires a correct (i.e. conforms to COMPO’s meta-model and semantics) way to reference it. Such a correct way is to have a required port connected to the default provided port of printingPort seen as a component (see Figure 4.) To achieve this, we have introduced the & operator for any port p, &p is such a required port. On our previous example, it is then possible to write &printingPort.isConnected() &printingPort is a primitive internal required port which is created on demand and automatically connected to the default port, itself a primitive port, of the printingPort port. Invocations sent through such a port are invocations sent to the component representing the printingPort port. An example of use of the & operator is given in the next paragraph. Because primitive ports are not reified, the application of the & operator on a primitive port returns itself, then a double application of the operator returns the same result as a single application, i.e. &printingPort == &&printingPort.

Example: A new kind of port - an aspect port

The following code snippet shows a toy integration of basic aspects to serve as an illustrating example. AspectPort defines in COMPO a new kind of required ports that have a special required port named aspectComp. As connected to any component having before and after services, let's call

---

8 A third one CollectionPort is not shown in the meta-model neither discussed in this section. reifying collection ports does not raise any additional issue.

9 As explained in Section 3.1 any component has a default provided port thought which all services the component owns are accessible.

---

Design and Implementation of a reflective Component-based programming and modeling language 2013/4/6
such a component an aspect component. It redefines the standard service invocation so that the before and after services of the aspect component are invoked before and after the standard invocation. The descriptor TextEditor shows a use of an aspect-port (note the ofKind statement to specify that an aspect required port is used). The connect statement in the architecture section of TextEditor descriptor says that the aspectComp port of the aspect required port &p, here used as a first-class component, should be connected to the default provided port of a MyAspectComponent, see Figure 5.

```
Descriptor AspectPort extends RequiredPort {
    requires { aspectComp : {before(); after(); }}
    service invoke(service) {
        aspectComp.before();
        super.invoke(service);
        aspectComp.after();
    }
}
```

```
Descriptor MyAspectComponent {
    provides { default : {before(); after()}}
    service before(){...}
    service after(){...}
}
```

```
Descriptor TextEditor {
    requires { printer ofKind AspectPort : {print()} }
    architecture {
        connect aspectComp@(&p)
        to default@(MyAspectComponent.new());
    ...
}
```

### Example: A new kind of port - a read-only port

The following code snippet shows the ReadOnlyProvidedPort descriptor realizing a new kind of provided ports through which only constant services, i.e. services not affecting the state of the component, could be invoked. It redefines the standard service invocation to check whenever it is correct or not to invoke the requested service and it also redefines the standard connecting service in a way, that a provided port of kind read-only can be delegated only to another read-only port.

```
Descriptor ReadOnlyProvidedPort extends ProvidedPort {
    service invoke(service) {
        [bool1 bool2]
        bool1 := owner.implements(service);
        bool2 := owner.isConstantService(service);
        if(bool1.and([bool2])]
        { super.invoke(service); }
        else { ... }
    }
    service connectTo(port) {
        if(port.getDescriptor().isKindOf(ReadOnlyPort))
        { super.connectTo(port); }
    }
}
```

To conclude this part on ports, we can say that their explicit status is a way to further control references between entities. For example, the above case of aspect required ports represents a way to realize a join point defined for all the users of a component having such a port. Or, the read-only example illustrates the fact that using different kinds of provided ports can facilitate different viewpoints on a component, in this case the read-only viewpoint.

### 3.3 First-class services

Listing 5 shows the COMPO implementation of the Service descriptor. Each service has a signature (port serviceSign to which an instance of ServiceSignature descriptor will be connected), temporary variables names and values (collection ports tempsN[] and tempsV[] ports), a program text (port code), actual parameters (collection port paramsV), an execution context (port context, to be connected at runtime to a component represent an execution context). For the sake of simplicity Listing 5 omits the architecture section and implementation of the execute() service. The service execute() checks if all requirements are satisfied, i.e. if a context component and components representing values of parameters are connected. Then it performs a system primitive to execute the code.

```
Descriptor Service extends Component {
    requires {
        context : IComponent;
        paramsV[] : *;
    }
    internally requires {
        serviceSign : ServiceSign;
        tempsN[] : IString;
        tempsV[] : *
        code : IString;
    }
    ...
    service execute() {...}
}
```

**Listing 5.** The Service descriptor.
The next section gives insights into the implementation that makes this meta-model and these definitions of COMPO descriptors in COMPO executable.

4. Bootstrap Implementation

![Diagram of Component and Descriptor](image)

**Figure 6.** Zoom in to the relation between Component and Descriptor descriptors

Although COMPO can be implemented in different languages, we have chosen Smalltalk, because its meta-model is extensible enough to support another meta-class system as shown in [4, 12].

Our meta-model is based on the two core concepts (captured in Figure 3): Component and Descriptor, the Figure 6 zoom in to their relation. Both are implemented as subclasses of Smalltalk-classes: Object and Class, respectively. Figure 7 shows their integration into the Smalltalk meta-model. This integration makes COMPO components and descriptors manageable inside Pharo Smalltalk environment. For example, one can use basic inspecting tool, the Inspector, Descriptor being defined as a sub-class of Smalltalk-class Class enables us to benefit from class management and maintenance capabilities provided by the environment. For example, all descriptors are “browsable” with the standard SystemBrowser tool.

One of the problems we challenged during the implementation is the fact that Smalltalk supports single-inheritance only. The meta-model shown in Figure 3 says that Descriptor inherits from Component, but as it is said above, we implement Descriptor as a sub-class of Smalltalk-classes (Class). Consequently Descriptor should have two parents and multiple-inheritance is needed. Concretely, there are two critical points, where multiple-inheritance is needed, marked with red ellipses in Figure 7: (i) Descriptor should inherit from Smalltalk-class Class and from Compo-class Component, to keep all benefits of Smalltalk’s classes management and in the same time to implement the meta-model design of COMPO; (ii) the automatically created Smalltalk-meta-class Component class should inherit from Smalltalk-meta-class Object class and from Compo-class Descriptor, to implement the fact that Component is an instance of Descriptor. To solve this we simulate the multiple inheritance by copy-

12 Although there is a solution based on single-inheritance, the solution introduces an issue when distinguishing components/descriptors from object classes in the implementation level.

![Integration of basic COMPO classes into Smalltalk meta-model](image)

**Figure 7.** Integration of basic COMPO classes into Smalltalk meta-model

Another problem we encountered is the implementation of Descriptor as an instance of itself. Smalltalk-class Descriptor is a unique instance of Smalltalk-meta-class Descriptor class, which is automatically created as a sub-class of Smalltalk-meta-class Class class (parallel hierarchy rule of Smalltalk) and therefore it does not have the same structure as Descriptor class. To solve this problem we have extended Smalltalk-meta-class Descriptor class in a way that it has the same attributes and provides the same methods as Smalltalk-class Descriptor.

Additionally, we have extented Object to behave as a primitive component providing all methods defined by Object (seen as compo services) through a unique provided port. Thus Smalltalk-objects are seen as primitive COMPO-components and they are usable in COMPO. This makes it possible to reuse Smalltalk class library. For example, the PrimitivePort Smalltalk-class can be used as rock-bottom primitive component used to implement primitive ports.

5. Application

Examples of introspection, intercession and meta-modeling applications have already been given in Section 3. Here we present two larger applications of these features, which were our main motivation to develop this work: a runtime component-based model transformation, and an architecture constraint checking.

The first application deals with a transformation scenario performed on COMPO’s implementation of the simple HTTP server, described in Section 2. This transformation migrates this component-based application from classic front-end/back-end architecture into a bus-oriented architecture. The transformation (sketched in Fig. 8) was motivated by a use-case when a customer (already running the server)
needs to turn the server with multiple front-ends and back-ends.

Figure 8. Simplified diagram illustrating the transformation from classic front-end back-end architecture into bus-oriented architecture.

To avoid the explosion of point-to-point connections we have decided to transform the original architecture into a bus-oriented one. A bus-oriented architecture reduces the number of point-to-point connections between communicating components. This, in turn, makes impact analysis for major software changes simpler and more straightforward. For example, it is easier to monitor for failure and misbehavior in highly complex systems and allows easier changing of components.

Listing 6. The ToBusTransformer descriptor.

```java
Descriptor ToBusTransformer {
  requires { context : IDescriptor }

  service stepOne - AddBus () {
    |pd cd|
    pd := PortDescription . new ('bus', 'required', 'internal', IBus);
    context . addPortDescription (pd);
    cd := ConnectionDescription . new ('bus', 'default@ (Bus . new ()) ');
    context . addConnectionDescription (cd);
  }

  service stepTwo - ConnectAllToBus () {
    |cd|
    cd := ConnectionDescription . new ('backEnd@fE', 'inputs@bus ');
    context . addConnectionDescription (cd);
    cd := ConnectionDescription . new ('default@bE', 'outputs@bus ');
    context . addConnectionDescription (cd);
  }

  service stepThree - RemOldConns () {
    |cd|
    cd := DisconnectionDescription . new ('backEnd@fE', 'default@bE ');
    context . removeConnectionDescription (cd);
  }
}
```

Listing 6. The ToBusTransformer descriptor.

The results of the transformation are checked using architecture constraints also implemented as COMPO components [32].

The transformation is modeled as a descriptor named ToBusTransformer. An instance was connected to the HTTPServer descriptor (COMPO’s code in Listing 1) and it performs the following transformation steps: (i) introduce a new internal required port named bus to which an instance of a Bus descriptor (not specified here) will be connected; (ii) extends the original architecture with new connections from front-end and back-end to bus; (iii) removes the original connection from front-end to back-end. Finally, a constraint component, an instance of the VerifyBusArch descriptor will be connected to the server to perform post-transformation verification. The constraint component executes a service verify which does the following steps: (i) verifies the presence of the bus component; (ii) verifies that the bus component has one input and one output port; (iii) verifies that all the other components are connected to the bus only and the original delegation connection is preserved.

Listings 6 and 7 show COMPO code of the ToBusTransformer descriptor and the VerifyBusArch descriptor. The following code snippet shows the use of the transformation and verification components:

```java
transformer := ToBusTransformer . new ();
constraint := VerifyBusArch . new ();

connect context@transformer to default@HTTPServer;
connect context@constraint to default@HTTPServer;

transformer . transform ();
constraint . verify ();
```

6. Related Work

In this section we compare reflection capabilities of COMPO with reflection capabilities provided by other component models. We classify the selected models into three categories: Modeling languages, Middleware component models and Component-based programming languages.

In the object-oriented world, two global approaches exist, when combining modeling and programming languages. The first one, takes a modeling language (such as EMOF) and integrates support for behavior description, i.e. programming support. For example KerMeta [20] explores how a meta-data language and a statically typed action language can be woven into a consistent executable meta-language. The second approach goes in the opposite direction, i.e. it extends programming languages with modeling languages features. Both approaches took advantage of maturity of support tools for their starting language, for example KerMeta is based on EMOF and therefore it is well-supported by modeling tools such as Eclipse/EMF; [12] is based on Smalltalk and therefore its programming facet is very well supported by Smalltalk’s environment (browser, inspector, debugger.)
In **COMPO** we apply the later approach in the component-oriented world.

**Modeling languages** UML 2 provides support for CBSE. UML itself is not a reflective language, but its meta-model (defined with MOF [21]) is. Reflection capabilities (manipulation of properties, invoke method, instance creation, etc.) provided by MOF are specifications only, i.e. there is no support for run-time reflection capabilities (as we introduced in **COMPO**).

A specific category of modeling languages are Architecture Description Languages (ADLs). The static nature of ADLs also do not match with reflection very well [16]. Reflection or at least introspection capabilities depend on code which is generated from architectures that these ADLs describe. For example, reflection is partially supported in C2 [17] through **context reflective interfaces**. Each C2 connector is capable of supporting arbitrary addition, removal, and reconnection of any number of C2 components.

**Middleware component models** Existing middleware technologies and standards provide very limited support for platform openness, usually restricted to high-level services, while the underlying platform is considered a black box. Recently, technologies such as interceptors, are a trend towards more openness. Nevertheless, the kind of openness provided is still limited to a few aspects of the platform.

CORBA Component Model (CCM) [23], Enterprise Java Beans (EJBs) [24] or Component Object Model (COM) [19] do not provide support for explicit architecture definition, the black-box approach they support does not fit with reflection very well. Introspection interfaces, which can be used to discover the capabilities of components, are the only reflection capability they offer. For example **CCM Navigation interface** for discovering the capabilities of components, are the only reflection capability they offer. For example **Cartoon Navigation interface** for discovering the capabilities of components, are the only reflection capability they offer. For example **CCM Navigation interface** for discovering facets (provided ports) or **IUnknown interface** in COM for discovering external (client and server) interfaces of a COM object. The interface **EJBContext** defines methods to retrieve references to the bean’s EJB home and remote interfaces classes, then normal Java reflection can be used to introspect the methods available to a client.

Only very few solutions consider reflection as a general approach which can be used as an overall framework that encompasses platform customization and dynamic reconfiguration. These models try to overcome the limitations of black-box approach by providing components with meta-information about their internal structure.


OpenCORBA is based on the meta-class approach and on the idea of modifying the behavior of a middleware service by replacing the meta-class of the class defining that service. This is mainly used to dynamically adapt the behavior of remote invocations, by applying the above idea to the classes of stubs and skeletons.

DynamicTAO is a CORBA compliant reflective ORB, which makes explicit the architectural structure of a system in a causally connected way. Component configurators keep the consistency of dependencies as new components are added or removed from the system. Reflection capabilities are limited to coarse-grained components, without possibility to control more detailed structures of the platform.

OpenCOM [7] (a lightweight and efficient component model based on COM) enables users to associate (dissociate) interceptor components with (from) some particular interface or to obtain all current connections between the host components’ receptacles and external interfaces.

Many reflection capabilities are supported in Fractal [5] component model, but the capabilities vary depending on kinds of Controllers (e.g. Attribute controller, Binding con-
controller, Content controller, ...) a Fractal component membrane contains. The Fractal specification provides several examples of useful forms of controllers, which can be combined and extended to yield components with different introspection and intercession features. An advanced example of using controllers is FRASCATI [29] model for development of highly configurable SCA solutions. In COPO, reflection capabilities are the same for all components (an orthogonal model). In addition, we go further in the reification of component-level concepts: services, ports and descriptors are components.

Furthermore, middleware component models are often designed to be platform independent. Then, for each platform, the tool support of these models generate code skeletons to be filled later. Consequently run-time transformations on components and their internal structure are performed through objects and not components. For example SOFA [25] reifies connectors. It is thus possible to specify high-level connectors within architecture descriptions. But finally, each primitive part of a connector specification has to be mapped to developers to some (object-oriented) code. Then reflection can be used if it is provided by this target (object-oriented) implementation language. In this case however, reflection do not address component-level concepts as in COPO.

Models@runtime [3] stream pushes the idea of reflection one step further by considering the reflection layer as a real model that can be uncoupled from the running architecture (e.g. for reasoning, validation, and simulation purposes) and later automatically re-synchronized with its running instance.

Meta-ORB [9, 26] proposes the design time use of models to generate middleware configurations, and, at runtime, the use of these same models as the causally connected self-representation of the middleware components that is maintained by the reactive meta-objects for the purposes of dynamic adaptation. Meta-ORB provides the meta-information management with a principled reflective meta-level. This has the benefit of unifying the use of meta-information in the system (e.g., preventing that different meta-object implementations use different meta-level representations), as well as providing a basis to closely integrate the configuration and adaptation features of the platform. In contrast to COPO’s orthogonal model where a change to a descriptor is propagated to all its instances, Meta-ORB reflection is based on per-object meta-objects, enabling to isolate the effects of reflection.

Kevoree is an open-source dynamic component model, which relies on models at runtime to properly support the dynamic adaptation of distributed systems. Kevoree introduces the Node concept to model the infrastructure topology and the Group concept to model semantics of inter node communication during synchronization of the reflection model among nodes. Kevoree includes a Channel concept to allow for multiple communication semantics between remote components deployed on heterogeneous nodes. All Kevoree concepts (Component, Channel, Node, Group) obey the object type design pattern to separate deployment artifacts from running artifacts. In opposite to COPO, where reflection capabilities are similar to all entities, Kevoree’s adaptation capabilities depend on different types of nodes (level 1 to 4).

The adaptation engine relies on a model comparison between two Kevoree models to compute a script for a safe system reconfiguration; execution of this script brings the system from its current configuration to the new selected configuration. Such adaptation scripts are written by designers, or they can be generated by automated processes (e.g. within a control loop managing the Kevoree system). In fact, the adaptation scripts are comparable to model transformations written in COPO.

The above described component models provide many sophisticated means for creating adaptable dynamic component-oriented solutions, but, in opposite to component-based programming languages like COPO, they use object-oriented programming to implement component-based software. Therefore there is no continuum to achieve the various stages of component-based software development using the same conceptual model.

**Component-based programming languages, CBPLs** The big advantage of CBPLs is that they do not separate architectures from implementation and so they have potential to manipulate reified concepts. In opposite to COPO, component-level concepts are often reified as objects, instead of components. This leads to a mixed use of component and object concepts. For example reflection package of ArchJava [1] specifies class (not component class) Port which represents a port instance. Very often the representations are not causally connected to concepts they represent. In case of ArchJava, which relies on Java reflection, the reason is that reflection in Java is mostly read-only, i.e. introspection support only.

Reflection is not explicitly advocated in ComponentJ [28]. It however appears that a running system certainly has a partial representation of itself to allow for dynamic reconfiguration of components internal architectures as described in [28] but it seems to be a localized and ad-hoc capability, the reification process being neither explicit nor generalized as in our proposal.

7. Conclusion

We have described an original operational reflective component-based programming language allowing for standard application development, and for static or runtime model and program transformations. Such a language offers a continuum to achieve the various stages of component-based software development in the same conceptual continuum. Such a continuum makes debugging or reverse-engineering simpler. It opens the essential possibility that ar-
chitectures, implementations and transformations can all be written at the component level and using a unique language. For example a programmer can design a component-oriented architecture, then verify the architecture’s properties and then seamlessly fill it in with code, all using COMPO. We have also given simple examples of how to write constraint checking (based on constraint components) and model transformations. As a reflective language giving a model access (via meta-components) to elements of the component-based meta-model, COMPO also makes it possible to design and implement new component-based construct (as exemplified with achieving a new kind of ports).

A key issue is uniformity, we have described a full component-based meta-model and a reflective description in COMPO of its main component descriptors made executable via a concrete implementation. We have proposed concrete, adapted (first-class descriptors) or new (first-class ports), meta-level solutions for a component-based reification of concepts leading to a “everything is a component” operational development paradigm.

COMPO in its today’s state is an operational prototype to develop complete component-based applications but is mainly conceived as a research laboratory to experiment with new ideas. To optimize programs efficiency is a remaining task but reflexivity is now well understood and many solutions do exist [6]. COMPO does not yet embed all new capabilities offered by existing ADLs or CBML, but its reflective architecture is especially designed to integrate them and to rapidly experience the impact of their integration. We thus have numerous perspectives in that direction such as to integrate first-class bound properties, aspects components, or more powerful solutions to express requirements and provisions.

Acknowledgments The authors would like to thank Roland Ducournau and Marianne Huchard for fruitful discussions.

References


