A Component-based meta-level architecture and prototypical implementation of a reflective Component-based Programming and Modeling language

Petr Spacek, Christophe Dony, Chouki Tibermacine

A Component-based meta-level architecture and prototypical implementation of a reflective Component-based Programming and Modeling language

Petr Spacek
LIRMM, CNRS and Montpellier II University
161, rue Ada, 34392 Montpellier Cedex 5 France
co-affiliation
Faculty of Information Technology
Czech Technical University in Prague
Thakurova 9 16000 Prague 6 Czech Republic
spacepe2@fit.cvut.cz

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

ABSTRACT

Component-based Software Engineering studies the design, development and maintenance of software constructed upon sets of connected components. Using existing standard solutions, component-based models are frequently transformed into non-component-based programs, most of the time object-oriented, for run-time execution. As a consequence many component-level descriptions (part of code), e.g. explicit architectures or ports declarations, vanish at the implementation stage, making debugging, transformations or reverse-engineering difficult. It has been shown that component-based programming languages contribute to bridge this gap between design and implementation and to provide a conceptual and practical continuum to fully develop applications with components. In this paper we go one step further in this direction by making a component-oriented programming and modeling language truly reflective, thus making verification, evolution or transformation stages of software development part of this new continuum. The gained reflection capabilities indeed make it possible to perform architecture checking, code refactoring, model transformations or even to implement new languages constructs with and for components. The paper presents an original executable meta-level architecture achieving the vision that "*everything is a component*" and an operational implementation demonstrating its feasibility and effectiveness. Our system revisits some standard solutions for reification in the component's context and also handles new cases, such as ports reification, to allow for runtime introspection and intercession on components and on their descriptors. We validate these ideas in the context of an executable prototypical and minimal component-based language, named Compo, whose first goal is to help imagining the future.

CBSE'14, June 30–July 4, 2014, Marcq-en-Baroeul, France.

http://dx.doi.org/10.1145/2602458.2602476 .

1. INTRODUCTION

Component-based software engineering studies the production of reusable components and their combination into connection architectures. It appears that component-orientation has been more studied from the design stage point of view, with modeling languages and ADLs [18, 10] than from the implementation stage one. 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 is somehow natural because object-oriented languages provide means to capture quite easily some of component-based concepts (encapsulation or provisions) and has practical advantages related to the availability and maturity of object-oriented programming languages, tools and practices. However this choice raises the issue that there is no conceptual continuum between models and their implementation (at least it is incomplete). Many concepts used during design (component descriptors, required services, ports, architectures, components themselves) vanish (are not represented as such, i.e. are not explicit) in the implementation.

This is a 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, 1]. 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. Some pieces of code, working at the meta level such as a constraint checking [32], may have to be written twice; once using elements of the design world meta-model and once using the implementation world meta-model. 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 *Reflect* package). Runtime model transformations, are only possible if the implementation language has an executable meta-model that allows for intercession; furthermore, after such a trans-

formation, a reverse-engineering is needed to update the model.

Facing these issues, component-based programming languages [1, 27, 13] propose a first global solution towards a continuum for component-based development. Modeling and programming languages [30, 31] extend the idea by making it possible to describe as well components, their connection architectures and their implementations in a unified component-based context. We aim at going further in this new work by suggesting that this continuum principle can also encompass all kinds of model-driven activities. This globally means to allow software engineers to define, using the same language described by a unique meta-model `M`, not only standard applications (architectures and code) having the same description of architecture at design-time and at run-time, 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[1].

We can rephrase this as defining a reflective modeling and programming component-based language (only structural reflection is considered). Two global approaches can be considered to achieve this goal : to extend a reflective modeling language with programming support (as done for example with KerMeta [21]), or to extend a reflective programming language with modeling support [12]. Both are interesting, the former provides a richer modeling context while the latter benefits from the efficiency of existing virtual machines. This paper describes a language named Compo, that applies the second approach in the component-based context, proposing an original "*everything is a component*" solution to build up an executable meta-model, allowing introspection and intercession. It can be used at all stages of component development to manipulate standard and "meta"-components as first-class entities. Various alternative solutions do exist for component-based application runtime adaptation (see the related works section) but we do not know of such a reflective solution being at the same time component-based, usable at all development stage and allowing for full intercession.

The paper is organized as follows. Section 2 presents Compo's standard syntax, constructs and use, necessary to the understanding of later examples. Section 3 presents several examples of Compo's reflection capabilities including constraints checking and model transformation. Section 4 presents the meta-model and how potential infinite regressions while interpreting it are solved. Section 5 describes Compo's prototype implementation in Smalltalk. Comparison with related works is presented in Section 6 and we conclude in Section 7 by discussing future work.

2. COMPO'S BASIC CONSTRUCTS

Our solution for reflection can be adapted to any component-based language. We present it in the context of our experimental solution Compo, a component-based language making services, provisions, requirements and internal architectures explicit. Thanks to this last point it is an ADL. As described in this section it is somehow a minimal ADL, but it is to be considered that the base presented here firstly is a partial view (see [29] for a complete presentation) and secondly should be augmented with newer solutions provided by advanced ADLs [18]. Before discussing and describing Compo's reflective version, it is needed that we give an overview of its basic constructs and syntax.

[1]Using such a solution of course does not imply to abandon those ideas brought by the component approach (e.g. the separation of development of architectures and implementations or transformation-based automatic deployment of components, etc). It simply opens the possibility that architectures, implementations and transformations can all be written at the component level and possibly (but not mandatorily) using a unique language.

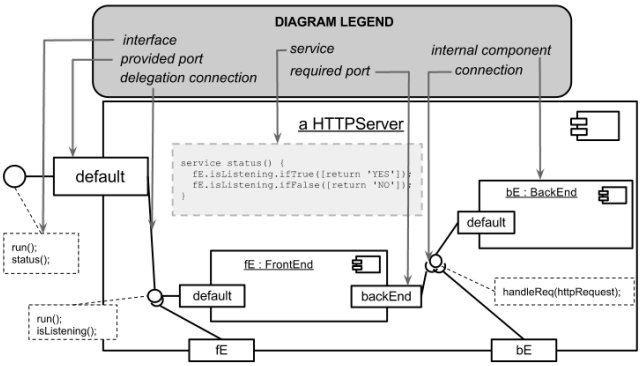

Figure 1: a component, instance of the HTTPServer descriptor

```
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.asstr() + ' is running'] }
         else { [return name.asstr() + ' is stopped'] };
    }
    service run() { ... }
}
```

Listing 1: An example of a component Descriptor : `HTTPServer`.

An excerpt of Compo component's model is described by the MOF meta-model in Figure 2 (only those concepts attributes and operations useful for this paper are shown in the figure and presented in the section). The language is based on a descriptor/instance dichotomy; components are instances of descriptors. Listing 1 shows an example of a descriptor named `HTTPServer` and Figure 1 an informal graphical representation of a component, instance of the `HTTPServer` descriptor. A descriptor describes the *structure* (ports and internal architecture descriptions) and *behavior* (service definitions) of its instances. For example, an `HTTPServer`'s behavior is defined with the `status` and `run` service definitions. External provided (vs. required) ports descriptions define the external *contract* (vs. *requirements*) of components. An `HTTPServer`'s external contract

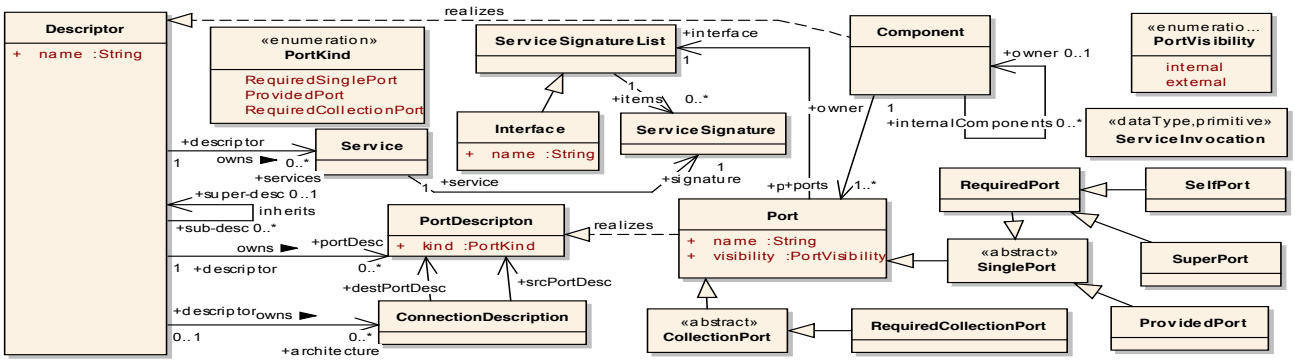

Figure 2: Compo's meta-model, before the integration of reflection

is to provide, as defined by its `default` provided port, the services `run` and `status`; and it has no external requirement[2]. A composite component has "*internal*" components, i.e. components to which it is connected via its internal required ports and unaccessible to the outside world. An HTTPServer is a composite with two internals, one instance of `FrontEnd` accessible via the internal required port `fE` and an instance of `BackEnd` accessible via `bE`. The architecture description of a composite define its *internal architecture* i.e. the way its internal components are accessible and how they are interconnected. The `architecture` of an `HTTPServer` is thus composed of an instance of `FrontEnd` connected to an instance of `BackEnd` via their ports as explained below.

Ports realize port descriptions (similarly to slots realizing classes' attributes in UML [23]). A port has a role (provided or required), a visibility (external or internal), a name and an interface. An interface is a list of service signatures which can be given either in extension (as for the `default` port of an `HTTPServer`) or via a descriptor name as a shortcut for the list of service signatures of its default provided port. Any component has at least 3 ports named: *default*, *self* and *super* (the semantics of these is explained in paragraph First-Class Component of Section 4.) Ports are connection 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). A connection establishes a communication channel between two ports. A regular connection connects a required port to a provided one, allowing for standard service invocation. An example of an expression establishing a regular connection is: `connect backEnd@fE to default@bE;` in `HTTPServer`'s architecture[3]. A delegation connection connects two ports having the same role and is used to delegate (or redirect) information from the outside to the inside of a component (or the opposite). An example of a "provided to provided" delegation connection is `delegate default@self to default@fE;` in `HTTPServer`'s architecture. Ports also are communication channels; any service invocation (e.g. `fE.isListening()` in service `status` in listing 1) is made via a required port and transmitted to the port it is connected to.

Finally Compo has an inheritance system [30] making it possible to define a descriptor as an extension of an existing one (`extends` keyword), to extend requirements and to extend or specialize architectures which has proven useful to our integration of reflection as it will be explained in the following sections (see the "inherits" reflexive association of `Descriptor` in Figure 2).

3. COMPO'S REFLECTION CAPABILITIES

In this section we present reflection capabilities of Compo. The MOF meta-model presented in Figure 3 shows Compo elements, representing the main component-level concepts, on which we apply the *component-oriented reification*[4]. 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 component-oriented reification means that for each element we create a descriptor, i.e. we define component-oriented representation of the elements, to turn these elements into first-class entities accessible in Compo's programs. Component-oriented reification supposes to solve various potential infinite regressions related to the definition of descriptors representing descriptors, ports and connectors (or connections). We detail these issues in the "Integrating reflection" section (Section 4.) In the following we present how these component-oriented representations can be used to achieve meta-programming, i.e. to design new kinds of descriptors and ports; to design and perform transformations and to verify architecture constraints.

An introspection example.

The following code snippet shows a basic use of introspection. The expression returns the descriptions of ports named `default`, `self` and `super`, which are defined by the descriptor `Component`, see Listing 4.

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

An intercession example.

The following code snippet shows the descriptor (named `ServiceMover`) of a refactoring component, which combines *get*, *remove* and *add* services to move a service from one descriptor to another.

[2]An example with an external requirement is given later.

[3]The expression `backEnd@fE` should be read: "the port `backEnd` of the component that will be connected to `fE` internal port of the current instance of `HTTPServer` (self)"

[4]This is an excerpt of the complete meta-model that only presents its central "interesting" parts.

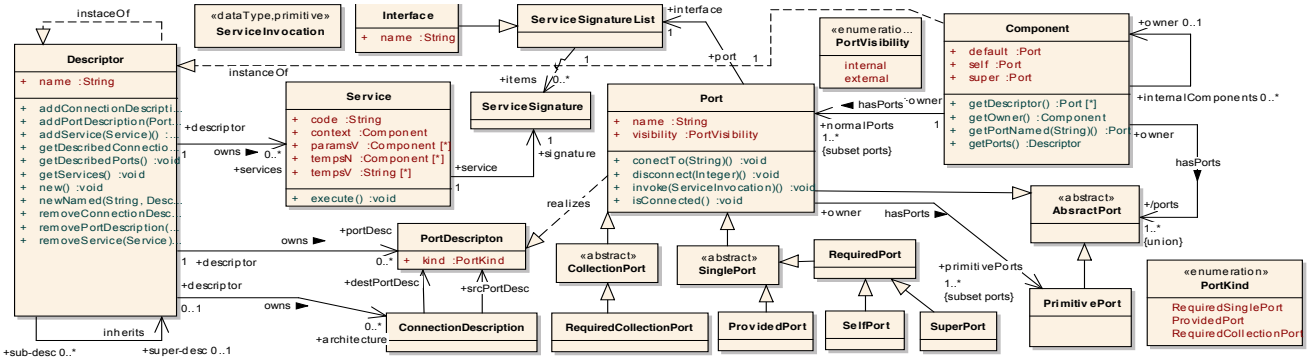

Figure 3: An Excerpt of the meta-model of Compo showing the integration of reflection

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

An example of defining a meta-descriptor.

`Descriptor` is a meta-descriptor, i.e. an entity whose instances are standard descriptors. A 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, the substitution of an instance of D by an instance of SD needs specific checking (child-parent incompatibility problem of inheritance systems in CBSE [30]). It may be wanted to define some descriptors that do not allow their sub-descriptors to add new requirements. Such a semantic restriction is achieved by the `DescriptorForSafeSubstitution` definition shown in the following code snippet. The meta-descriptor extends the meta-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 (reg & 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
{ ... }
```

An Example of 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 services without side effect, 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 provided 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 read-only example illustrates the fact that using different kinds of provided ports can facilitate different view-points on a component, in this case the read-only view-point.

An Example of a transformation design and constraint verification.

Examples of introspection, intercession and meta-modeling applications have already been given in previous paragraphs. Here we present two larger applications[5] 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. 4) was motivated by a use-case when a customer (already running the server) needs to turn the server with multiple fronts-ends and back-ends.

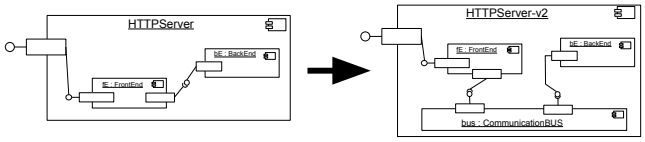

Figure 4: Simplified diagram illustrating the transformation from classic front-end back-end architecture into bus-oriented architecture.

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. This makes easier monitoring for failure and misbehavior in highly complex systems, and allows easier modifications on components.

```
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() {...}
 service stepThree-RemOldConns() {...}
}
```

Listing 2: A code-snippet of 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.

[5]Actually we present code snippets because of the space limit.

```
Descriptor VerifyBusArch extends Constraint
{
 service verify() {...}
 service stepOne-IsBusPresent() {...}
 service stepTwo-HasBusIOPorts(busPD){...}
 service stepThree-AreAllConnsToBus(busPD){
  |conns|
  conns := context.getConnsDescs();
  conns.remove([:cd|cd.getSrcPort()
                  .getInterface()==IBus]);

  if((conns.remove([:cd|
      cd.isDelegation()])))
  {} else { return false };

  if(conns.forEach([:cd|
   (cd.srcPortDesc()==busPD)
   .or([cd.destPortDesc()==busPD])
  ]) {return true } else { return false };
 }
}
```

Listing 3: A code-snippet of the `VerifyBusArch` descriptor.

Listings 2 and 3 show snippets of COMPO code of the `ToBusTransformer` descriptor and the `VerifyBusArch` descriptor. The following code snippet shows the use of the transformation and verification components:

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

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

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

4. INTEGRATING REFLECTION

This section describes the integration of structural reflection[6] capabilities into COMPO, 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]. The integration is based on the new version of the meta-model presented in Figure 3. The following sub-sections explain it, present the COMPO's reflective description of its essential elements, discuss the issues it eventually raises and give clues on how the associated interpretation processes, that makes the meta-model executable, cope, when necessary, with infinite regressions potentially induced by cycles it contains.

First-Class Components.

[6]Our solution makes it possible to define new kind of ports (e.g. aspect ports, see [29]) in which service invocation can be altered, this is a very limited form of behavioral reflection; we offer no way to control or modify basic services invocation and execution.

Our "*everything is a component*" requirement is achieved via the transposition of the Smalltalk [14] original solution (also re-introduced for the same purpose in MOF reflection [22]) that any entity is an instance of a descriptor and that *Component* (conceptually conforming to `MOF::Reflection::Object`) is the root of the descriptor inheritance hierarchy; this makes, together with the classical set theoretic interpretation of inheritance, any instance of any descriptor a component.

The `Component` descriptor then defines the basic structure and behavior shared by all components. Its reflective definition in COMPO (cf. Listing 4) shows that any component has at least one provided port named `default` through which all public services it defines (`*` notation) can be invoked. Any component also has two internal provided ports named `self` and `super`[7] allowing a component to send service invocations to itself via these ports. `Component` also defines different services of global interest, like `getDescriptor()` which returns the receiver's descriptor.

```
Descriptor Component {
  provides { default : * }
  internally requires {
    super : * ofKind SuperPort;
    self : * ofKind SelfPort;
  }
  service getPorts() {...}
  service getPortNamed(name) {...}
  service getDescriptor() {...}
  service getOwner() {...}
}
```

Listing 4: The `Component` descriptor.

First-class descriptors.

Components as we see and propose them with COMPO differ from objects mainly by their explicit requirements, explicit architectures, their connections and their communication via ports[8]. Concerning the question of seeing and manipulating descriptors as components, these differences do not prevent from reusing standard meta-classes solutions to manipulate classes as objects. Among existing solutions, we have chosen the ObjVlisp one [8], because (i) it solves in the simplest way the problem that each component, including descriptors, should be an instance of a descriptor, and (ii) it is perfectly adapted to our requirements of allowing for new explicit meta-descriptors (e.g. independent from the descriptors inheritance hierarchy.

Translated in our context, it leads to the COMPO's recursive definition given in Listing 5 where `Descriptor` is instance of itself and extends `Component`. Of course it needs a bootstrapped implementation to become executable. This is presented in Section 5. It defines the `new` service to create instances and `newNamed(name, super-desc)` to create new descriptors and some base services for using 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. In addition to what is defined in `Component`, a descriptor has four internal required ports: `name`, `ports` (a collection of instances of `PortDescription` according to which ports of its instances are), `architecture` (a collection of instances of `ConnectionDescription`[9] representing its instances architecture description), and `services` (its services dictionary).

```
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 getDescribedConnections() {...}
  service addService(service) {...}
  ...
}
```

Listing 5: The `Descriptor` descriptor.

First-class ports.

Unlike the case for descriptors, seeing and manipulating ports as components raises an interesting open issue. Ports are high-level abstractions to represent connections and the capacity for architects to disconnect and reconnect other components, whatever form they can take, either possibly physical connections of physical devices or references to addresses in memory. Reifying ports resembles to handling first-class references [2], but looks at them at a higher abstraction level. This also allows do deal with the problem in a simple way (and obviously without handling efficiency considerations). Ports abstract connection points and communication channels. Reifying them opens the door to the creation of various kinds of connections or to various ways to send, transmit, interpret or control services invocations (read-only ports described in section 3 is one example, aspect-ports would be another one (see [29]).

The problem that we should tackle in this case is the following: Reified ports should have ports: i) internal required ports to hold some internal data, for example their name or the name of the interface that describe them, ii) one external provided ports to which other components could connect to invoke their services. Using the base level, it should be possible to use ports in a standard way as a communication channel to transmit services invocations as for example with the expression `printer.print('hello')` (see Figure 5), where `printer` is a port of a component `c`, will invoke the service

[7]The statement `ofKind` in the definition states that the `self` and `super` ports are created as instances of specific descriptors `SelfPort` and `SuperPort` respectively.

[8]This vision is compatible with other components models, including those that consider components as packaging entities. If it were to be considered how to pack one COMPO component and put it on a shelves, as for javabeans, we would consider putting in the pack its descriptor, or its descriptor with this particular instance, or its descriptor together with all the descriptors of its internal components, etc.

[9]`PortDescription`s and `ConnectionDescription`s are used at instantiation time and also during static or runtime architecture checking or transformation. In the case of a runtime transformation implementation it should be ensured that descriptor level descriptions and instances internal representations are still causally connected, when the description changes.

print of the component connected to c via printer. Using the meta-level, it should be possible to use ports as standard components, i.e. for example to invoke, in a way that conforms to COMPO's meta-model and semantics, the isConnected() service (see Listing 6) for port printer, which should return true. The above requirements induce an infinite regression as soon as it is tried to invoke a service of a port seen as a component. To build a usable system, we have altered them by denying the component status to ports of ports. This alters in a very marginal way the reflective possibilities of the system. All examples presented in the previous section, and all applications we can think of, stay achievable.

Our complete solution comes as follows. All standard ports can be used as components on demand. All Standard ports conform to the COMPO reflective definition of the Port descriptor shown in listing 6. A primitive port is not a component and cannot be used as such. Any port of any standard port is automatically created as a primitive port by the virtual machine. The attachment of a primitive port to its port is primitive and done by the virtual machine. The special & operator (the Semantics of which bears some resemblance with the C & operator's one) is introduced such that, for any port p, &p is, and behaves as such, a primitive internal required port automatically connected to the default port (all components have it), itself a primitive port, of p. Used in the context of our previous example, it is then possible to write &printer.isConnected(), where &printer is a port of the current component connected to a primitive port of printer. The invocation is then transmitted to printer seen as a component thus invoking the isConnected() service of Port descriptor. Because primitive ports are not components, they cannot be used as such and the& operator is made idempotent, &printer == &&printer, etc.

It is to be noted that to have ports made explicit is a way to abstract connections between components. Having first class ports also opens the door to introspection and experimentation with various kinds of connections [19]. The read-only ports defined in section 3 can also be seen as implementing read-only connectors. Together with this capability our model makes it possible to define adapter components and interconnect them between any components.

```
Descriptor Port extends Component {
  requires {
    owner : IComponent
    connectedPorts[] : IPort
  }
  internally requires {
    name : IString;
    interface : IInterface
  }
  service getName(){...}
  service getIterface(){...}
  service invoke(service){...}
  service isConnected(){...}
  service connectTo(port){...}
  service disconnect(index){...}
}
```

Listing 6: The Port descriptor.

First-class services.

Reifying services as components does not raise new issues compared to their reification as objects, and it is shortly

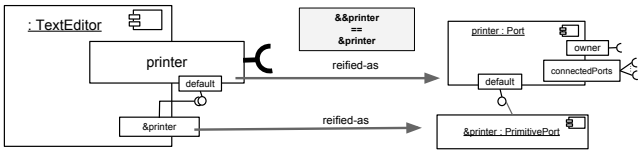

Figure 5: The & operator for accessing the component-oriented reification of the printer port of component TextEditor

given to complete the presentation. Listing 7 shows the COMPO implementation of the Service descriptor. Each service has a signature (port serviceSign) to which an instance of ServiceSignature[10] descriptor will be connected), temporary variables names and values (collection ports tempsN and tempsV), a program text (port code), actual parameters (collection port paramsV), an execution context (port context, to be connected at run-time to a component representing an execution context). To shorten, the architecture section and implementation of the execute() service are omitted.

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

Listing 7: The Service descriptor.

Next section gives insights into the implementation that makes this meta-model and these COMPO definitions of COMPO descriptors executable.

5. BOOTSTRAP IMPLEMENTATION

COMPO could be implemented from scratch, e.g. with a new virtual machine, but our first solution to validate our ideas more rapidly has been to implement it on top of an existing language having objects and a reflective level offering intercession on structures. Among various possible alternatives (CLOS could have been used), we have chosen Smalltalk, because it has been shown that its meta-model is extensible enough [4, 12] to support another meta-class system.

Our meta-model is based on the two core concepts (captured in Figure 3): Component and Descriptor. Both are implemented as sub-classes 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

[10]Due to space reasons we omit the ServiceSignature descriptor definition. To give a hint, it provides a service to store and access names and parameters names of services.

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. We benefit from such a deep integration and provide Compo also its own tool to support descriptors' design process, see Figure 6. By the way, the tool is also an example of reflection usage.

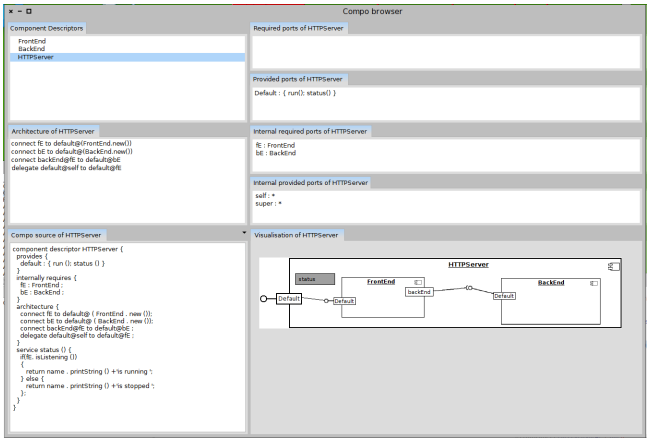

Figure 6: Screenshot of the Compo's HTTPServer implementation in the descriptor's development tool.

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[11]. 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` (ii) the automatically created Smalltalk-meta-class `Component class` should inherit from Smalltalk-meta-class `Object class` and from Compo-class `Descriptor`. To solve this we simulate the multiple inheritance by automated copying attributes and methods from `Component` to `Descriptor` and from `Object class` to `Component class`, when one of the parents evolves.

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 can be seen as primitive Compo-components and they are usable in Compo. This

[11]Although there is a solution based on single-inheritance, the solution introduces a unsuitable issue when distinguishing components/descriptors from objects/classes in the implementation level.

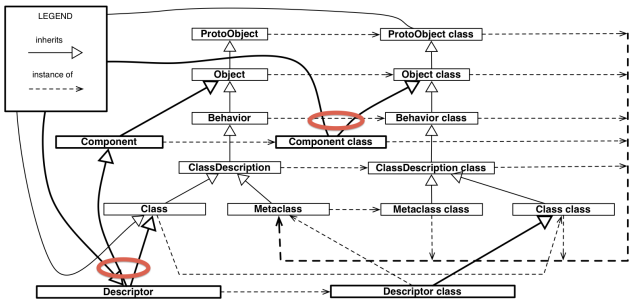

Figure 7: Integration of basic Compo classes into Smalltalk meta-model

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.

6. RELATED WORK

In this section we compare reflection capabilities and solutions offered by Compo with those provided by other component-based systems. For that purpose we classify existing systems into three categories: Modeling languages, Middleware component models and Component-oriented programming languages.

Modeling languages.

UML 2 provides support for component-based software modeling. Although UML itself is not a reflective language, its meta-model (defined with MOF [22]) is. Reflection capabilities (manipulation of properties, 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 well with reflection [18]. Reflection or at least introspection capabilities depend on the 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 limited support for platform openness, usually restricted to high-level services, while the underlying platform is considered as a black box.

CORBA Component Model (CCM) [24], Enterprise Java Beans (EJBs) [25] or Component Object Model (COM) [20] do not provide support for explicit architecture definition. The black-box approach they support does not fit with reflection very well. Introspection interfaces (like `IUnknown` interface in COM), which can be used to discover the capabilities of components, are the only reflection capability they offer.

Only few solutions consider reflection as a general approach which can be used as an overall framework that encompasses platform customization and dynamic recon-

figuration. These models try to overcome the limitations of the black-box approach by providing components with meta-information about their internal structure. Projects like OpenCOM [7] (a lightweight and efficient component model based on COM), OpenCORBA [16] and DynamicTAO [15] adopt reflection as a principled way to build flexible middleware platforms. In opposite to Compo, reflection capabilities of these are usually limited to coarse-grained components, without the possibility to control more detailed structures of platforms.

Many reflection capabilities are supported in Fractal component model [5], but the capabilities vary depending on kinds of *Controllers* a Fractal component membrane contains. Controllers can be combined and extended to yield components with different introspection and intercession features as shown by FRASCATI model [28] for the development of highly configurable SCA solutions. In Compo, 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 by providing tools for generating code skeletons to be filled later. Consequently, and in opposite to Compo, run-time transformations on components and their internal structure are performed through objects and not components. as it is for example in SOFA [26] and its reification of connectors which has to be mapped by developers to some (object-oriented) code.

Meta-ORB [9] as a representant of the Models@runtime stream [3] 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. In contrast to Compo'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.

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 Compo, 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 a class (not a component class) named `Port` which represents a port instance. 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 [27]. It however appears that a running system certainly has a partial representation of itself to allow for dynamic reconfiguration of internal architectures of components as described in [27] but it seems to be a localized and ad-hoc capability, the reification process being neither explicited nor generalized as in our proposal.

7. CONCLUSION

We have described an operational original reflective modeling and programming component-based language allowing for the development of component-based models (architectural part at this point), applications and of static or runtime model and program transformations. It offers developers an effective conceptual continuum to use components at all stages of software development with no conceptual loss between stages. It opens the essential possibility that architectures, implementations and transformations can all be written at the component level and using a unique language. As a reflective language giving a model access (via meta-components) to elements of a uniform 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).

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 some concrete, adapted from existing works (first-class descriptors) or new (first-class ports), solutions for a component-based reification of concepts leading to a uniform "*everything is a component*" operational development paradigm.

Compo in its today's state is an operational prototype mainly designed as a research laboratory to experiment new ideas. Here are the main ones we intend to follow on. (1) Compo does not yet embed all new capabilities offered by existing ADLs and modeling languages, but its reflexive architecture is especially designed to integrate them and to rapidly experience the impact of their integration. (2) Using the same reflective capabilities, we aim at integrating generalized bound properties (primarily exemplified by JavaBeans), aspects components, an more powerful solutions to express requirements and provisions. (3) The global idea that objects plus explicit requirements and explicit architectures are components is already largely present in this paper, we aim at experiment its full generalization. (4) To improve Compo's program efficiency can be done in various ways; firstly known optimization for reflexive language [6] but following previous point 3 we can imagine updating existing virtual machines for reflective object-oriented languages to components-based ones.

Acknowlegments.

The authors would like to thank Roland Ducournau, Luc Fabresse and Marianne Huchard for fruitful discussions.

8. REFERENCES

[1] J. Aldrich, C. Chambers, and D. Notkin. Archjava: connecting software architecture to implementation. In *Proceedings of the 24th International Conference on Software Engineering*, ICSE '02, pages 187–197, New York, NY, USA, 2002. ACM.

[2] J.-B. Arnaud, M. Denker, S. Ducasse, D. Pollet, A. Bergel, and M. Suen. Read-only execution for dynamic languages. In *Proceedings of the 48th international conference on Objects, models, components, patterns*, TOOLS'10, pages 117–136, Berlin, Heidelberg, 2010. Springer-Verlag.

[3] G. Blair, N. Bencomo, and R. France. Models@ run.time. *Computer*, 42(10):22–27, 2009.

[4] J.-P. Briot and P. Cointe. Programming with explicit metaclasses in smalltalk-80. *SIGPLAN Not.*, 24(10):419–431, Sept. 1989.

[5] 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.*, 36(11-12):1257–1284, Sept. 2006.

[6] S. Chiba. Implementation techniques for efficient reflective languages. Technical report, Departement of Information Science, The University of Tokyo, 1997.

[7] M. Clarke, G. S. Blair, G. Coulson, and N. Parlavantzas. An efficient component model for the construction of adaptive middleware. In *Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms Heidelberg*, Middleware '01, pages 160–178, London, UK, UK, 2001. Springer-Verlag.

[8] P. Cointe. Metaclasses are first class: The objvlisp model. *SIGPLAN Not.*, 22(12):156–162, Dec. 1987.

[9] F. M. Costa, L. L. Provensi, and F. F. Vaz. Using runtime models to unify and structure the handling of meta-information in reflective middleware. In *Proceedings of the 2006 international conference on Models in software engineering*, MoDELS'06, pages 232–241, Berlin, Heidelberg, 2006. Springer-Verlag.

[10] I. Crnkovic, S. Sentilles, A. Vulgarakis, and M. Chaudron. A classification framework for software component models. *Software Engineering, IEEE Transactions on*, 37(5):593 –615, sept.-oct. 2011.

[11] F.-N. Demers and J. Malenfant. Reflection in logic, functional and object-oriented programming: a short comparative study. In *In IJCAI '95 Workshop on Reflection and Metalevel Architectures and their Applications in AI*, pages 29–38, 1995.

[12] S. Ducasse and T. Gîrba. Using smalltalk as a reflective executable meta-language. In *Proceedings of the 9th international conference on Model Driven Engineering Languages and Systems*, MoDELS'06, pages 604–618, Berlin, Heidelberg, 2006. Springer-Verlag.

[13] L. Fabresse, N. Bouraqadi, C. Dony, and M. Huchard. A language to bridge the gap between component-based design and implementation. *COMLAN : Journal on Computer Languages, Systems and Structures*, 38(1):29–43, Apr. 2012.

[14] A. Goldberg and D. Robson. *Smalltalk-80: The Language and Its Implementation.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1983.

[15] F. Kon, M. Román, P. Liu, J. Mao, T. Yamane, C. Magalhã, and R. H. Campbell. Monitoring, security, and dynamic configuration with the dynamictao reflective orb. In *IFIP/ACM International Conference on Distributed systems platforms*, Middleware '00, pages 121–143, Secaucus, NJ, USA, 2000. Springer-Verlag New York, Inc.

[16] T. Ledoux. Opencorba: A reflective open broker. In *Proceedings of the Second International Conference on Meta-Level Architectures and Reflection*, Reflection '99, pages 197–214, London, UK, UK, 1999. Springer-Verlag.

[17] N. Medvidovic, P. Oreizy, J. E. Robbins, and R. N. Taylor. Using object-oriented typing to support architectural design in the c2 style. *SIGSOFT Softw. Eng. Notes*, 21(6):24–32, Oct. 1996.

[18] N. Medvidovic and R. N. Taylor. A classification and comparison framework for software architecture description languages. *IEEE Trans. Softw. Eng.*, 26(1):70–93, Jan. 2000.

[19] N. R. Mehta, N. Medvidovic, and S. Phadke. Towards a taxonomy of software connectors. In *Proceedings of the 22nd international conference on Software engineering*, ICSE '00, pages 178–187, New York, NY, USA, 2000. ACM.

[20] Microsoft. *COM: Component Object Model Technologies.* Microsoft, 2012.

[21] P.-A. Muller, F. Fleurey, and J.-M. Jézéquel. Weaving executability into object-oriented meta-languages. In *Proceedings of the 8th international conference on Model Driven Engineering Languages and Systems*, MoDELS'05, Berlin, Heidelberg, 2005. Springer-Verlag.

[22] OMG. *Meta Object Facility (MOF) Core Specification Version 2.4.1*, 2011.

[23] OMG. *Unified Modeling Language (UML), V2.4.1.* OMG, August 2011.

[24] OMG. *CORBA Component Model (CCM).* OMG, 2012.

[25] Oracle. *Enterprise JavaBeans Specification Version 3.* Oracle, 2012.

[26] 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.

[27] J. C. Seco, R. Silva, and M. Piriquito. Componentj: A component-based programming language with dynamic reconfiguration. *Computer Science and Information Systems*, 05(02):65–86, 12 2008.

[28] L. Seinturier, P. Merle, R. Rouvoy, D. Romero, V. Schiavoni, and J.-B. Stefani. A component-based middleware platform for reconfigurable service-oriented architectures. *Softw. Pract. Exper.*, 42(5):559–583, May 2012.

[29] P. Spacek. *Design and Implementation of a Reflective Component-Oriented Programming and Modeling Language.* PhD thesis, Montpellier II University, Montpellier, France, December 2013.

[30] P. Spacek, C. Dony, C. Tibermacine, and L. Fabresse. An inheritance system for structural & behavioral reuse in component-based software programming. In *Proceedings of the 11th GPCE*, pages 60–69. ACM, 2012.

[31] P. Spacek, C. Dony, C. Tibermacine, and L. Fabresse. Wringing out objects for programming and modeling component-based systems. In *procs. of the 2nd Int. Workshop on Combined Object-Oriented Modeling and Programming Languages (COOMPL'13) - co-located with ECOOP.* ACM Digital Library, July 2013.

[32] C. Tibermacine, S. Sadou, C. Dony, and L. Fabresse. Component-based specification of software architecture constraints. In *Proceedings of the 14th CBSE*, pages 31–40, New York, NY, USA, 2011. ACM.