Wringing out objects for programming and modeling component-based systems

Petr Spacek, Christophe Dony, Chouki Tibermacine, Luc Fabresse

Wringing out Objects for Programming and Modeling Component-based Systems

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

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

ABSTRACT

Languages and technologies used to implement component-based software are not component-based, i.e. while the design phase happens in the component world, the programming phase occurs in the object-oriented world. When an object-oriented language is used for the programming stage, then the original component-based design vanish, because component concepts like requirements and architectures are not treated explicitly. This makes it difficult to keep model and its implementation causally connected. We present a new pure reflective component-based programming and modeling language where all core component concepts are treated explicitly and therefore keeping original component-design alive. The language makes it possible to model and program component-based software using the same language plus its uniform component-based meta-model and integrated reflection capabilities make the language and its applications open and flexible.

Categories and Subject Descriptors

D.1 [**Programming techniques**]; D.2.11 [**Software Architectures**]: Languages; D.3 [**Programming languages**]

General Terms

Languages, Programming, Modeling

Keywords

programming, modeling, metamodeling, reflection, inheritance, requirements, architecture, component

1. INTRODUCTION

Nowadays trend is to develop complex software by assembling reusable software pieces called components. The core component concepts: *explicit external contract* and *explicit internal composition* facilitate modeling phase of software design process. For these purposes, many component oriented frameworks and models exist and provide tools and languages [6, 12] making them suitable for formal architecture design, reasoning and manipulation.

However, programming phase facilitation is mainly focused on generating code skeletons into target programming languages, which are very often object-oriented. In other words, while the design phase happens in the component world, the programming phase occurs in the object-oriented world. In our work we address this by proposition of a pure component-based language, which combines modeling capabilities of Architecture Description Languages (ADLs) [12] with implementation powers of standard programming languages.

Although object-oriented languages provide conceptual means for understanding and organizing knowledge about phenomena and concepts from the real-world (application domain, in our case component-based software), only a small percentage of the final code relates to the real world, i.e. modeling and programming coupling disappears. When an object-oriented language is used, then the original component-based design may vanish in the programming stage, because component concepts like requirements and architectures are treated implicitly. Consider the following example where we use Java to model a very simple text-editor component.

```
class TextEditor {
  private ISpellChecker sc;
  public TextEditor() {  }
  public void setSpellChecker(ISpellChecker sc) {...}
  public ISpellChecker getSpellChecker() {...}
  ...
}
```

The global semantics of the `sc` attribute with the getter and setter operations is: “a text-editor requires a spell-checker”. Unless users of such an editor read the editor’s documentation or its code, they are not aware of the fact that the editor requires a spell-checker. The information is not explicit.

Moreover, component architectures described in current component-oriented frameworks and models are not easily verifiable and transformable. The solutions do not offer capability to manipulate model entities at run-time and therefore architecture constraints and transformations has to be defined in 3rd party languages [7, 19]. Only few solutions

inspired by models@run.time philosophy [3], provide a reflective solution with a casual connection between a model and its run-time representation, but even if they do so, the run-time manipulation is object-oriented.

In our work we focus on modeling and programming component-based systems. We believe that component systems should be implemented in languages where components have first-class status and are able to explicitly express what they offer (*provisions*) and what they need (*requirements*), in order to implement desired behavior. They also should be able to explicitly describe internal composition, called *architecture*, i.e. to be composed of other components (internal components or sub-components.)

Like in [10] or UML, we also see *components* as just special objects able to explicitly describe provisions, requirements and architectures. Components' contracts are defined in terms of ports, which are connection and communication units each described by a name, an interface, a role (provided or required) and a visibility (external or internal). Provisions of a component are defined via provided ports and conceptually they resemble interfaces an object implement. Requirements of a component are defined via required ports and conceptually they are related to attributes. External required ports resemble public (or accessible via getters and setters) attributes. Internal required ports resemble protected and private attributes. An architecture is a system of internal components and of connections between them. It resembles assignments to private and protected attributes in the constructor method of an object.

In this work we present a reflective component-based language named Compo making it possible to model and program component-based software using the one language. Our proposal has the following original contributions:

- Components are seen as extended 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 by revealing the original design intentions and therefore helping to understand.
- It applies 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.
- Its reflection capabilities make it possible to develop standard component-based application, to perform advanced architecture checking, code refactoring or model transformations, all using the same language.

The rest of this paper is organized as follows. Section 2 presents Compo's standard syntax, constructs and use when modeling and programming component-oriented software; Section 3 presents advanced example of use: a model transformation verified by constraints components. Section 4 describes Compo implementation. Comparison with related works is presented in Section 5 and we conclude in Section 6 by discussing future work.

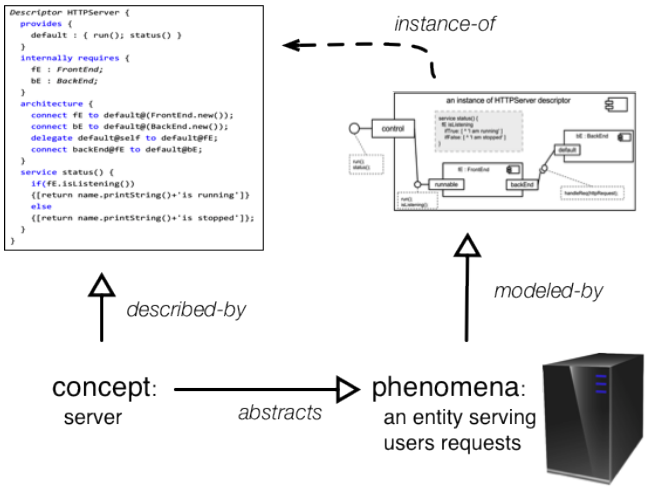

Figure 1: The relations between terms: phenomena, concept, descriptor and component

2. THE LANGUAGE

In the following we present Compo's programming and modeling capabilities on a walk-through example in which we design and implement a very simple HTTP server.

Modeling stage. The HTTP server concept is an abstraction of the real world phenomena of an entity serving clients' requests. Compo component model is based on a descriptor/instance dichotomy where components are instances of descriptors (similarly to object/class dichotomy.) As shown in Figure 1, a descriptor describes a concept and its instances are models of the phenomena the concept abstracts.

```
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());
    ...
    connect backEnd@fE to default@bE;
  }
  ...
}
```

Listing 1: The `HTTPServer` descriptor - modeling stage.

At a glance, the Listing 1 shows a definition of a descriptor named `HTTPServer` describing 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[1]. These internal components are connected together so that the front-end can invoke services of the back-end.

In fact, at this moment (prior the programming stage) the HTTPServer descriptor is fully comparable to component definitions in ADLs [12, 17] and it could be used as a regular architecture description to generate a code skeleton into a programming language. In this sense, Compo is an ADL, but as you will see in the following, one of our contributions is that the behavior of the description can be seamlessly implemented without the need to switch into another language or technology.

Programming stage. To program the run and status services the server provides, we complete the HTTPServer descriptor with the implementation of the status service and with the delegation connection from the default port of the server to the provided port default of the front-end. The delegation states that the run service is implemented by the front-end. The Listing 2 shows the completion of the descriptor.

Figure 2 shows a diagram of the HTTPServer descriptor's instance after modeling and programming have been finished.

Descriptors. Let's look at each point more precisely. A descriptor defines the *structure* and *behavior* of its instances. The behavior is given by a set of services 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 .

```
Descriptor HTTPServer {
  provides { ... }
  internally requires { ... }
  architecture {
    ...
    delegate default@self to default@fE;
    ...
  }
  service status() {
    if(fE.isListening())
    { return name.printString()
                +'is running';
    } else {
      return name.printString()
                +'is stopped';
    };
  }
}
```

Listing 2: The HTTPServer descriptor - programming stage.

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.

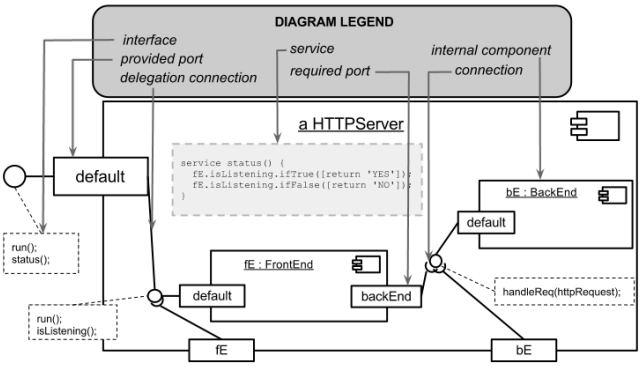

Figure 2: Diagram of an HTTPServer component instance

An important feature of Compo is inheritance. The integrated inheritance system[2] [18] is quite innovative in the context of CBSE, because it promotes modeling power with covariant specializations and presents an approach to deal with the child-parent incompatibility problem of inheritance systems in CBSE. A descriptor can be defined as a sub-descriptor of an existing one using the extends statement, this gave us ability to reuse descriptors and to model hierarchies of descriptors. Sub-descriptors inherit all parts defined by their super-descriptor and can extend or specialize them. For example, a sub-descriptor can introduce new services and its instances can access and reuse services defined by its super-descriptor. This gives us ability to define behavior that's specific to a particular sub-descriptor, i.e. achieve polymorphism of descriptors.

Our modeling scheme follows the idea of "*everything is a component*" and represents descriptors as components. It is directly inspired from [5] and conforms to the MOF solution [14]. *Component* is our root classifier, that conforms to MOF::Reflection::Object. *Descriptor* is our basic meta-classifier, that conforms to UML::Classes::Kernel::Classifier. *Descriptor* is the descriptor of descriptors, i.e. a meta-descriptor and 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 Component and Descriptor descriptors[3], the implementations of *Component* and *Descriptor* respectively.

[1]For the sake of simplicity we do not show the FrontEnd and BackEnd descriptors.

[2]Single inheritance type

[3]Due to space reasons, we omit their Compo implementations

Descriptors `Component` and `Descriptor` define the basic introspection and intercession behavior making it possible to observe and adapt components dynamically. Moreover, with the inheritance system, it is possible to define new kinds of meta-descriptors, i.e. to achieve meta-programming and meta-modeling.

Ports. Ports are connection and communication points. A service invocation is made via a required port and transmitted to the provided port, the required port is connected to. `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. A port has a role (provided or required), a visibility (external or internal), a name and an interface, optionally the `kindOf` statement can be use to specify the descriptor of the port, we explain this later. Ports are instances of `Port` descriptor[4] and they are true components. This is important for model checking and transformations and also to allow for defining new kind of ports introducing new communications protocols, e.g. revocable or read-only ports as hinted in Listing 3. 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 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 also primitive.

```
Descriptor ReadOnlyPort extends Port {
  service invoke(service) {
    ...
    if(owner.isConstantService(service))
    { return super.invoke(service) };
    ...
  }
  ...
}
```

Listing 3: The `ReadOnlyPort` descriptor

Connections. Connections 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. An example of an expression establishing a regular connection is: `connect backEnd@fE to default@bE;`[5], (see. Listing 1) . As a support for 1:N relationships we introduce *collection ports*. A required (resp. provided) port can be declared as a collection port (syntax is `<portName>[]`) meaning that the port can be connected to one or more provided (resp. required) ports accessible through an index. Collection ports boots modeling power of the language making it possible to model dynamic architectures, for example a component with variable number of internal components. In opposite to ports or descriptors, connections do not have first-class status in Compo. 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 [13] because, using inheritance and `kindOf` statement, our model makes it possible to define and use new kind of ports; and because of the capability it offers to put an adapter component in between any components.

Services. Services implement the behavior of components and semantically they conform to methods in OOP. Each service has a signature, temporary variables names and values, a program text, actual parameters and an execution context. They are defined in descriptors using the `service` statement, for example the `status` service in Listing 2. Refactoring operations (add, remove, move), run-time behavior modification, JIT compilation and other features are possible when services do have first-class status. Services are instances of `Service` descriptor which describes their structure, i.e. it reifies formal parts of services like name, parameters, etc. Compo does not provide behavior reflection, which focus on reification of concepts from which behavior description is composed, i.e. assigments, invocations, because it may led to inefficient programs, as shown in [11].

3. ADVANCED EXAMPLE

In this example we model and perform a transformation which migrates the HTTP server component-based application (presented in Section 2) from classic front-end/back-end architecture into a bus-oriented architecture. The transformation (sketched in Fig. 3) was motivated by a use-case when a customer (already running the server) needs to turn the server into a server with multiple fronts-ends and backends.

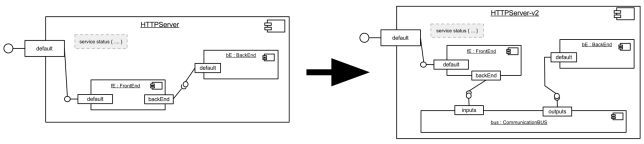

Figure 3: Simplified diagram illustrating the transformation from classic front-end back-end architecture into bus-oriented architecture.

The bus-oriented architecture reduces the number of point-to-point connections between communicating components, which 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.

The results of the transformation are checked using architecture constraints also implemented as Compo compo-

[4]Its Compo implementation is omitted due to space reasons

[5]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.

```
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 4: The `ToBusTransformer` descriptor.

nents [20].

The transformation is modeled as a descriptor named `ToBusTransformer`. An instance was connected to the `HTTPServer` descriptor 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.

Listing 4 gives a hint of COMPO code of the `ToBusTransformer` descriptor. Due to space reasons we omit the code of 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. PROTOTYPE IMPLEMENTATION

A prototype of our language is implemented in Pharo Smalltalk[6]. We have chosen this model, because it is extensible enough to support another meta-class system as shown in [8]. Our meta-model is based on the two core concepts: `Component` and `Descriptor`. Both are implemented as sub-classes of Smalltalk-classes: `Object` and `Class`, respectively. 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. COMPO also comes with its own tool to support descriptors' modeling process, see Figure 4.

[6]www.pharo-project.org

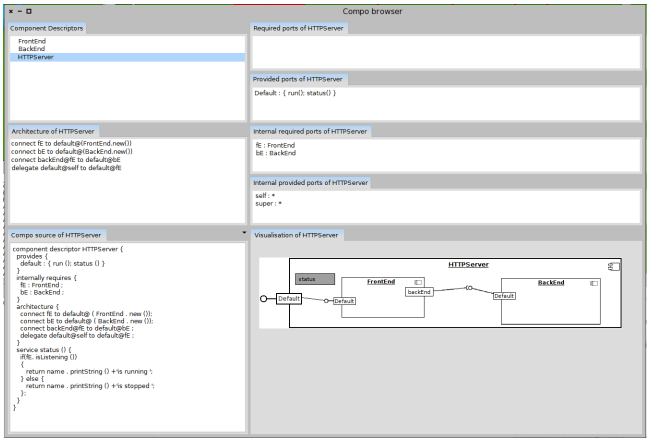

Figure 4: Screenshot of the Compo's HTTPServer implementation in the descriptor's development tool.

5. RELATED WORK

The big advantage of Component-based Programming Languages (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 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.

Reification and reflection are not explicitly advocated in ComponentJ [16]. However, it appears that a running system certainly has a partial representation of itself to allow for dynamic reconfiguration of components internal architectures as described in [16] but it seems to be a localized and ad.hoc capability, the reification process being neither explicited nor generalized as in our proposal. This makes architecture reasoning, constraints and transformations difficult to implement. ComponentJ favor composition over inheritance as a reuse mechanism. This has advantage of not introducing additional language-level mechanism, but makes it difficult to reuse formal structure definitions and achieve hierarchies modeling.

SCL [9], the predecessor of COMPO, is a uniform CBPL implemented in Smalltalk. COMPO shares with SCL many features like unique communication protocol, unplanned connections support or services' arguments passing. With respect to SCL, COMPO goes further in modeling aspect, its explicit architectures support, meta-model and inheritance

system boots modeling power of the language and provide basis for Model Driven Development.

CLIC [4], an extension of Smalltalk to support full-fledged components, which provides component features such as ports, attributes, or architecture. In opposite to COMPO, the model is not implementation independent, it focus on symbiosis between CLIC and Smalltalk plus it enables to benefit from modularity and reusability of components without sacrifice performance. CLIC fully relies on Smalltalk reflective capabilities, its components are objects and their descriptors are extended Smalltalk classes. Compared to COMPO, modeling powers of CLIC are limited, the model allows components to have only one provided port. The authors argue that it is hard to split component functionality over multiple ports, because developers do not know beforehand, which services will be specified by each required port of client component.

6. CONCLUSION

We have described an original operational reflective component-based programming language allowing for standard component-based application development by supporting modeling and programming development phases. Such a language offers a continuum to achieve the various stages of component-based software development in the same conceptual continuum. The continuum makes debugging or reverse-engineering simpler. It opens the essential possibility that architectures, 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. Or, in the spirit of MDE, several transformations can be modeled and implemented to enhance primary models into final products. Moreover, COMPO's reflexive architecture allows to experience the impact of adding new mechanisms at both the architectural and implementation levels.

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

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

[6] 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.

[7] K. Czarnecki and S. Helsen. Feature-based survey of model transformation approaches. *IBM Syst. J.*, 45(3):621–645, July 2006.

[8] 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.

[9] 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.

[10] O. L. Madsen and B. Møller-Pedersen. A unified approach to modeling and programming. In *Proceedings of the 13th international conference on Model driven engineering languages and systems: Part I*, MODELS'10, pages 1–15, Berlin, Heidelberg, 2010. Springer-Verlag.

[11] J. Malenfant, C. Dony, and P. Cointe. Behavioral reflection in a prototype-based language. In *Proceedings of International Workshop on Reflection and Meta-Level Architectures*, pages 143–153. ACM, 1992.

[12] 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.

[13] 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.

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

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

[16] 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.

[17] 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.

[18] 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.

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

[20] 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.