Rapid Prototyping of Domain-Specific Architecture Languages

Romain Rouvoy, Philippe Merle

Rapid Prototyping of Domain-Specific Architecture Languages

Romain Rouvoy
University Lille 1, LIFL CNRS UMR 8022
Inria Lille – Nord Europe, France
romain.rouvoy@univ-lille1.fr

Philippe Merle
Inria Lille – Nord Europe, France
University Lille 1, LIFL CNRS UMR 8022
philippe.merle@inria.fr

ABSTRACT

Software architecture has become a sensitive discipline, which consists in concretizing the user requirements into a set of artifacts that can be used to model and reason about the software to be developed. However, the architect often relies on its own knowledge to map domain-specific requirements onto generic software abstractions. Most of the time, this leads to the definition of repetitive tasks and architecture fragments, which can be particularly error prone. We therefore believe that architects need a more flexible approach to cope with the definition of domain-specific architectures by leveraging general purpose architecture description languages. This paper introduces the FraSCAla framework as an adaptive architectural framework that can be used to rapidly prototype and experiment domain-specific ADLs in order to catalyze the definition and to improve the reliability of software architectures. We demonstrate the merits of this approach on two representative architectural patterns of component-based software architectures.

Categories and Subject Descriptors

D.2.11 [**Software Architectures**]: [Domain-specific architectures, Languages, Patterns]

Keywords

language prototyping, component-based architectures

1. INTRODUCTION

Software architecture has become a sensitive discipline, which consists in concretizing the user requirements into a set of artifacts that can be used to model and reason about the software to be developed. This task becomes even more critical when the complexity of the developed software increases or when the software addresses a specific domain. In both cases, the architect relies on its own knowledge to map domain-specific requirements onto generic software abstractions, like components. However, this process can lead to the definition of repetitive tasks and architecture fragments, which can be particularly error prone. Therefore, software architects are rather tempted to manipulate domain-specific concepts which could be seamlessly converted to technical artifacts. Nonetheless, the definition of such domain-specific languages is a tedious process, which requires a lot of experience from the architect to identify the relevant constructions and a lot of time to develop the appropriate toolchain.

We therefore believe that architects need a more flexible approach to cope with the definition of domain-specific architectures by leveraging general purpose *Architecture Description Languages* (ADLs). In this paper, we therefore introduce the FraSCAla framework as an adaptive architectural framework that can be used to rapidly prototype and experiment domain-specific ADLs in order to catalyze the definition and to improve the reliability of software architectures—*i.e.*, FraSCAla is a framework for building "à la carte" ADLs. In particular, we promote a layered approach (as depicted in Figure 1) to isolate the definition of the structural architectural model from the language statements used to describe software architectures. We demonstrate the merits of this approach on two architectural patterns that are defined by the FraSCAti middleware platform [21] and the COSMOS context framework [19].

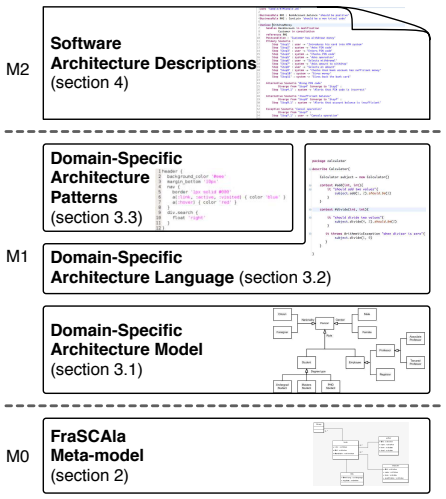

Figure 1: Overview of the contribution.

The remainder of this paper is therefore organized as follows. We first introduce the FraSCAla framework, which groups the essential concepts of our contribution (cf. Section 2). Then, we build on this kernel to demonstrate the definition of an ADL for the *Service Component Architec-*

ture (SCA) industrial standard (cf. Section 3). The resulting ADL is assessed by the illustration of two case studies which exploit FraSCAla to describe complex software architecture that exhibit architectural patterns (cf. Section 4). In the last part of the paper, we compare our contribution with the state-of-the-art (cf. Section 5) before concluding and sketching some of the perspectives for this work (cf. Section 6).

2. THE FRASCALA FRAMEWORK

FraSCAla is a framework for prototyping the definition of *Architectural Description Languages* (ADL). The objective of FraSCAla is to leverage the construction of ADLs by providing an homogeneous environment to introduce new architectural concepts according to domain-specific requirements. Although it mostly targets prototyping activities, FraSCAla promotes a structured approach to build ADLs by adopting an incremental design approach. First, FraSCAla defines a *meta-model* to introduce the essential concepts required to prototype an ADL. Then, instances of this meta-model are used to describe categories of ADL (*e.g.*, component-based ADLs). Each FraSCAla *model* can be further refined to fit the specificities of a given component model (*e.g.*, *Service Component Architecture* standard [2]). Once a target model is defined, FraSCAla introduces the associated *language* by constraining this model with a concrete syntax. This concrete syntax introduces the keywords of the ADL as well as the associated operations it supports for describing software architectures. The last feature of FraSCAla consists in supporting the definition of *architectural patterns*. Software architecture patterns are a powerful construction for isolating reusable fragments of software architecture descriptions. By enabling the definition of architectural patterns, FraSCAla captures domain-specific architectures as advanced patterns. To reduce the burden of using the complex MDE technologies like the *Eclipse Modelling Framework* (EMF) [22], FraSCAla builds on the Scala programming language [17, 18]. In particular, FraSCAla benefits from Scala's support for type inference, genericity, implicit definition, trait, closures, as well as native XML support. The remainder of this section therefore describes the core of FraSCAla and how it can be used to sketch an ADL for component-based systems.

2.1 Defining an ADL Meta-Model

Figure 2 describes the core meta-model of FraSCAla, which essentially builds on two entities: Concept and Controller. In particular, a Concept is used to introduce a new architectural element in the model, and is structured as a composition of Controllers. A Controller can be considered as a mixin [5] isolating a specific concern of the Concept. The identification of Controllers fosters their reuse by several Concepts and it enables a Concept to be assembled "*à la carte*".

Both a Concept or a Controller can define a set of Parameters, which can refer to a primitive Value (supported by the enumeration DataType) or provide a Reference to another architectural Entity.

This meta-model composes the conceptual kernel of FraSCAla and can be used to prototype any ADL. The following section illustrates how this conceptual kernel can be used to define a generic ADL for describing component-based systems.

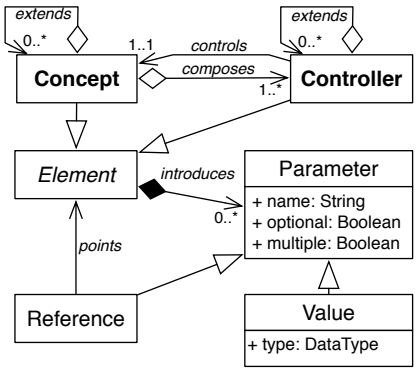

Figure 2: FraSCAla Meta-Model.

2.2 Deriving a CBSE Model

Software architectures have already been adopted by various Software Engineering communities [4]. In the context of this paper, we focus on the definition of an ADL conforming to the principles of *Component-Based Software Engineering* [8]. In particular, we demonstrate that FraSCAla can be used to describe a generic component model, which can be incrementally refined towards a more concrete component model and ultimately a domain-specific component model. Figure 3 depicts the generic component model we designed using the stereotypes introduced by the FraSCAla meta-model. This model is strongly inspired from the Fractal component model, as described in [5]. Our model is structured around the concept of Membrane, basically defined as a composition of a Name and an Annotation controller. A Membrane is an empty software cell extended as a Component Membrane, which includes the Port and Property controllers. One can observe that both the Membrane and the Gate concepts (inherited by the Port and Property concepts) include the Annotation controller. Annotations provide a variability point in this generic model in order to decorate a Membrane or a Gate with specific metadata. SubMembrane and Composite Membrane are themselves defined as a Membrane extension integrating specific controllers.

The definition of this yet another generic component model provides an architectural pivot, which can be used to automatically instrument concrete component models (*e.g.*, introspection, verification).

2.3 Implementing the Model in Scala

Model-Driven Engineering technologies like the *Eclipse Modelling Framework* (EMF) [22] provide advanced frameworks for describing meta-models, models as well as their graphical or textual syntaxes. However, these frameworks tend to suffer from their intrinsic complexity and do not provide much elasticity to support the definition of modular ADLs. In particular, the definition of a new concept usually requires to generate the associated classes and to modify the concrete syntaxes in order to support this concept. However, such invasive modifications conflict with our objective to build ADLs "*à la carte*" by composing architectural concepts on-demand. Furthermore, the definition of EMF models, as well as concrete syntaxes, introduces a steep learning curve, which is not expected when prototyping architectural constructions. Another direction could be to encode FraSCAla models with an executable meta-modelling language,

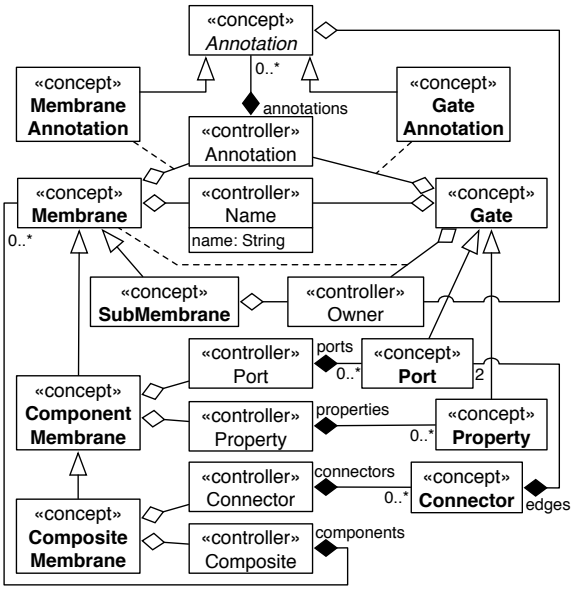

Figure 3: Component-based ADL Model.

such as KERMETA[1]. Nonetheless, extending the KERMETA language with embedded DSL constructions seems to be currently a complex engineering task. For these reasons, we investigated another approach, which consists in exploiting an advanced general-purpose programming language like Scala not only to implement the FRASCALA models, but also to provide a textual syntax and to describe software architectures. This reflective approach provides an homogeneous environment to quickly define and exploit new software architectural constructions.

Scala is a multi-paradigm programming language combining features of object-oriented programming and functional programming [17, 18]. Scala supports functional programming by providing a lightweight syntax for defining *anonymous functions*, supports *higher-order functions*, allows functions to be *nested*, and supports *currying*. Scala is also a pure object-oriented language in the sense that every value is an object. Data types and behaviors of objects are described by *classes* and *traits*. Class abstractions are extended by subclassing and by a flexible *mixin-based composition mechanism* to avoid the problems of multiple inheritance. Scala is able to infer types by usage, thus the lack of explicit type declarations gives Scala the appearance of a dynamically typed language. Scala acknowledges that the development of domain-specific applications often requires domain-specific language extensions. New language constructs are added in the form of *libraries*, which facilitates the definition of new statements without extending the syntax nor using macro-like meta-programming facilities.

Listing 1[2] therefore reports a Scala code excerpt of three of the CBSE controllers we defined in the previous section. Concretely, each Controller defined in the model is encoded as a `trait`, which is a construction provided by Scala to isolate a specific concern (cf. lines 1, 5 & 10). Traits map a Parameter in the model to a variable (using the keyword `var`, cf. lines 2, 7 & 12), while generic concepts can be mapped to an abstract type (using the keyword `type`, cf. lines 6 & 11). Besides, parameters of type `T` with a cardinality multiple are mapped to a Scala `Set[T]` (cf. line 12), while those exhibiting a contingency optional are converted to the class `Option[T]` (illustrated in Listing 4).

```
1  trait NameController {
2    var name: String = _
3  }
4
5  trait OwnerController {
6    type OWNER
7    var owner: OWNER = _
8  }
9
10 trait AnnotationController {
11   type ANNOTATION <: Annotation
12   var annotations = Set[ANNOTATION]()
13 }
```

Listing 1: Examples of CBSE Controllers.

Similarly, Listing 2 reports on the implementation of CBSE concepts into Scala. A Concept in the model consists in a `trait` in Scala, which mixes the required set of controllers (using the keywords `extends` and `with`). A concept can further constrain the types introduced by the inherited controllers using the primitive `type`. For example, the `PortAnnotation` (cf. lines 4–6) specializes the `AnnotationController` specification by imposing that the owner of the annotation should be a subtype of the concept `Port` (introduced in lines 8–11). The component-based concepts are therefore assembled from controllers, which isolates crosscutting concerns as reusable entities (*e.g.*, `NameController` or `AnnotationController`).

```
1  trait Gate extends NameController with OwnerController
2   with AnnotationController
3
4  trait PortAnnotation extends Annotation {
5    type OWNER <: Port
6  }
7
8  trait Port extends Gate {
9    type ANNOTATION <: PortAnnotation
10   type OWNER <: PortController
11 }
12
13 trait MembraneAnnotation extends Annotation {
14   type OWNER <: Membrane
15 }
16
17 trait Membrane extends NameController
18  with AnnotationController {
19   type ANNOTATION <: MembraneAnnotation
20 }
21
22 trait ComponentMembrane extends PortController
23  with PropertyController
```

Listing 2: Examples of CBSE Concepts.

The resulting Scala implementation defines the foundations of a component model, which remains agnostic from any technology. In the remainder of this paper, we illustrate how this model is further refined and instantiated to prototype an ADL for a specific technology.

3. SUPPORTING THE SCA STANDARD

CBSE has been recognized as a suitable methodology for building modular systems deployable in a variety of environments ranging from wireless sensor networks [23] to complex

[1] `http://www.kermeta.org`

[2] In all listings, texts in red are Scala or FRASCALA keywords, texts in blue are FRASCALA type and variable identifiers, while texts in green are string constants.

systems-of-systems [10, 15]. This recognition has resulted in the definition of several academic [5, 7, 12] or industrial [2] specifications aiming at establishing a standard for CBSE. However, the multiplication of these standards does not help in capitalizing supporting tools, but rather requires to invest in the development of specific solutions. In this section, we therefore illustrate that, using FraSCAla, developers can easily specialize a generic component model into a technology specific one, while capitalizing on the supporting tools.

3.1 Mapping to the SCA Specification

The *Service Component Architecture* (SCA) [2, 21] is a set of OASIS's specifications for building distributed applications based on *Service-Oriented Architecture* (SOA) and CBSE principles. The key benefit is that SCA is independent from programming languages, *Interface Definition Languages* (IDL), communication protocols, and non-functional properties. As illustrated in Figure 4, in SCA the basic construction blocks are software *components*, which have *services* (or provided interfaces), *references* (or required interfaces) and expose *properties*. The references and services are connected by means of *wires*. SCA specifies a hierarchical component model, which means that components can be implemented either by primitive language entities or by subcomponents. In the latter case the components are called *composites*. To support interactions via different communication protocols, SCA provides the notion of *binding*. For SCA references, a binding describes the access mechanism used to invoke a service. In the case of services, a binding describes the access mechanism that clients use to execute the service. Each SCA concept has a standardized graphical representation (as shown in Figure 4) and could be instantiated via a concrete XML-based syntax. In this paper, we therefore show how FraSCAla could help to build a concrete ADL for SCA and providing mapping facilities to dump a FraSCAla definition into a standard SCA XML-based descriptor.

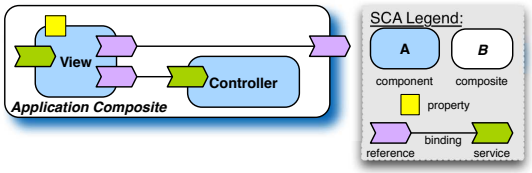

Figure 4: Representation of an SCA Architecture.

The code excerpt below reflects part of the configuration depicted in Figure 4 using the SCA assembly language:

```
<composite name="MyApp"                                          1
           xmlns="http://www.osoa.org/xmlns/sca/1.0">            2
  <service name="run" promote="View/run"/>                       3
  <component name="View">                                        4
    <implementation.java class="app.gui.SwingGuiImpl"/>          5
    <service name="run">                                         6
      <interface.java interface="java.lang.Runnable"/>           7
    </service>                                                   8
    <reference name="model" autowire="true">                     9
      <interface.java interface="app.ModelService"/>             10
    </reference>                                                 11
    <property name="orientation">landscape</property>            12
  </component>                                                   13
  <!-- ... -->                                                  14
</composite>                                                     15
```

Listing 3: Description of an SCA application.

Although XML-based description supports extensions to be defined, the definition of such extensions is bound to variability points allowed by the core schema. Furthermore, XML-based descriptions are purely declarative, which requires an appropriate assembly engine to parse and process the definition to implement the semantics of the model. In this paper, we therefore show how we can achieve a similar expressivity by using an embedded DSL to leverage current limitations of declarative ADLs. We also show that our approach remains compatible with these declarative ADLs by providing mapping facilities to dump a FraSCAla definition into a standard SCA one.

Figure 5 provides a graphical representation of the SCA model in FraSCAla. This model consists in refining the concepts of the component model we introduced in Section 2.2 with the domain-specific concepts imposed by SCA. For example, the concepts Service and Reference of SCA are indirectly mapped (via the abstract concept Contract) to the concept of Port in the generic component model, while the SCA concepts of Interface and Binding are reified as PortAnnotation. In SCA, both interface and binding definitions can be considered as metadata that decorates the definition of a port. Furthermore, the FraSCAla framework supports the injection of new concepts in the mapping like the controller Intent whose semantics has no equivalence in the generic component model.

The encoding of this SCA model in Scala is then reported in Listing 4. Interestingly, one can observe that the traits `ScaInterface` and `ScaBinding` are sealing the type `OWNER` with `ScaContract` (cf. lines 7 & 12). Line 17 illustrates the definition of an optional parameter `interface` in the controller `ScaInterfaceController`. Finally, the concept `ScaContract` consists in the refinement of the concept `Port` with the SCA-specific controllers `ScaInterfaceController`, `ScaBindingController`, `ScaIntentController`, and `XmlController`. While the controllers `ScaInterfaceController`, `ScaBindingController`, and `ScaIntentController` are used to store the interface and all the bindings and intents associated to the contract—*i.e.*, a service or a reference—the controller `XmlController` provides a function to generate the equivalent definition using the XML notation promoted by the SCA standard. This function is used to ensure the compatibility with the SCA standard and deploy software architectures described with this language on reference implementations of the SCA specifications, such as OW2 FraSCAti[3], Fabric3[4], IBM WebSphere[5], or Apache Tuscany[6].

```
trait XmlController {                                            1
  def toXML: scala.xml.Node                                      2
}                                                                3

trait ScaInterface extends PortAnnotation                        5
 with XmlController {                                            6
  type OWNER = ScaContract                                       7
}                                                                8

trait ScaBinding extends PortAnnotation                          10
 with XmlController {                                            11
  type OWNER = ScaContract                                       12
}                                                                13

trait ScaInterfaceController                                     15
 extends AnnotationController {                                  16
```

[3] http://frascati.ow2.org
[4] http://www.fabric3.org
[5] http://www.ibm.com/software/websphere
[6] http://tuscany.apache.org

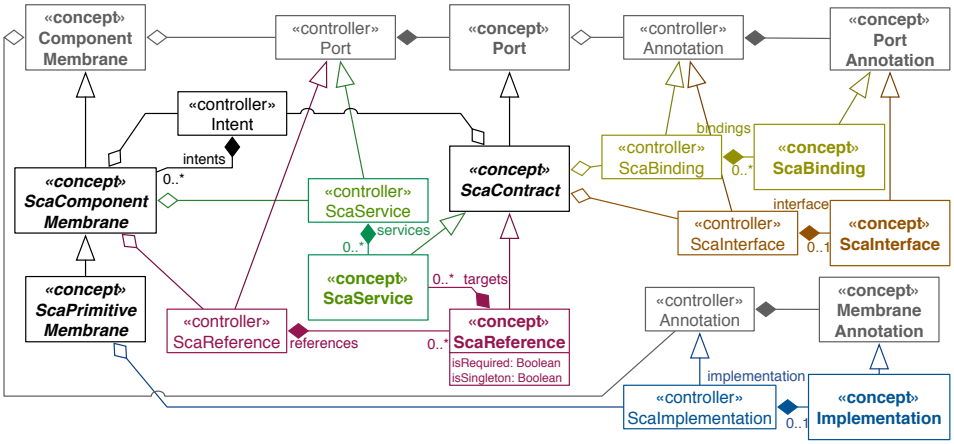

Figure 5: SCA Mapping.

```
  var interface: Option[ScaInterface] = None
}

trait ScaBindingController
 extends AnnotationController {
  val bindings = new Set[ScaBinding]()
}

trait ScaContract extends Port with XmlController
 with ScaInterfaceController with ScaBindingController{
  type OWNER <: ScaComponentMembrane
  type ANNOTATION = PortAnnotation
}
```

Listing 4: SCA Model.

Similarly, Listing 5 reports on the implementation of the variability points defined by the SCA standard. In SCA, various technologies can be used to describe an interface, to expose a binding, or to implement a component. Therefore, the above introduced concepts of `ScaInterface`, `ScaBinding`, and `ScaImplementation` are variability points of our SCA model in order to leverage the support of additional technologies. Such extensions are integrated in our framework by providing an implementation of the associated trait (cf. Listing 5). Additionally, each extension describes also its mapping to the XML notation supported by the SCA standard by implementing the function `toXML` of `XmlController`. Interestingly, while the use of *case classes* (cf. lines 1, 6 & 11) hides the definition of new instances, the classes `Java` and `Bean` benefit from the concept of `Manifest` provided by the Scala library to overcome Java's type erasure (cf. lines 1 & 6). Manifests are therefore used to manipulate a reification of the type parameter and automatically resolved by the Scala runtime using the keyword `implicit`. This approach provides a convenient notation for our prototype language as well as typing checking capabilities.

```
case class Java[T](implicit var itf: Manifest[T])
   extends ScaInterface {
  def toXML = <interface.java interface={ itf }/>
}

case class Bean[T](implicit var cls: Manifest[T])
   extends ScaImplementation {
  def toXML = <implementation.java class={ cls }/>
}

case class HTTP(var uri: String) extends ScaBinding {
  def toXML = <http:binding.rest uri={ uri }
  xmlns:http="http://frascati.ow2.org/xmlns/sca/1.1"/>
}
```

Listing 5: SCA Variability.

Discussion. This first mechanism allows the architect to easily introduce new technologies in the definition of SCA-based architectures. Indeed, beyond the technologies specified by the SCA standard, the reference implementations of SCA, such as OW2 FraSCAti or Apache Tuscany, provide a support for additional technologies, such as HTTP (as illustrated in Listing 5), script languages (*e.g.*, Python, Groovy, JavaScript) or interface definition languages (*e.g.*, UPnP, WADL, OMG IDL). By extending the appropriate traits, the architect can therefore adapt the FraSCAla framework to include these extensions.

3.2 Building the SCA Language

Once the structural model is defined, the FraSCAla framework introduces the syntax to be used for describing software architectures. Similarly to the principle of architecture refinement, the syntax is introduced into the model by refining the concepts with the associated statements. Listing 6 provides an illustration of this process. The trait `scaComponent` introduces an abstract statement by extending the concept `ScaComponentMembrane` defined at the model level. This statement constrains the relationship between the concepts of the model by introducing the appropriate scopes. For example, `scaComponent` defines the sub-types `scaContract` (cf. lines 6–34) and `scaProperty` (cf. lines 36–42), which inherits from the concepts `ScaContract` and `ScaProperty`, respectively. This means that a contract and a property can only be defined within the scope of a component. This subtyping relationship provides access to the attributes of the enclosing type to automatically register their references (cf. lines 9, 12, 18, 27, 33 & 49). The introduced functions are used to hide the model construction primitive using domain-specific verbs (*e.g.*, `exposes`, `is`). We also exploit the design pattern *builder* (cf. lines 11, 17 & 39) to chain function calls when describing a component. We believe that this notation provides a natural syntax for describing component-based SCA architectures.

```
trait scaComponent extends ScaComponentMembrane {
  type OWNER <: scaComposite
  type PORT = scaContract
  type PROPERTY <: scaProperty[_]
```

```
  trait scaContract extends ScaContract {
    type OWNER <: scaComponent
    type ANNOTATION <: PortAnnotation
    ports += scaContract.this

    def exposes[I <: ScaInterface](i: I): this.type ={
      annotations += i
      interface = Some(i)
      i.owner = this
      this
    }
    def as[B <: ScaBinding](b: B): this.type = {
      annotations += b
      bindings += b
      b.owner = this
      this
  } }
  case class service(name: String)
   extends ScaService with scaContract {
    type OWNER = scaComponent
    owner = scaComponent.this
    services += this
  }
  case class reference(name: String)
   extends ScaReference with scaContract {
    type OWNER = scaComponent
    owner = scaComponent.this
    services += this
  }

  trait scaProperty[T] extends ScaProperty[T] {
    type OWNER <: scaComponent

    def is(v: T): this.type = {
      value = v
      this
} } }
class Composite extends ScaCompositeMembrane
 with scaComponent {
  case class component(name: String)
   extends ScaPrimitiveMembrane with scaComponent {
    type OWNER = Composite
    owner = Composite.this
    components += this
} }
```

Listing 6: SCA Language.

Listing 7 reports on a self-contained example of a component-based HelloWorld application. The first part of the description (cf. lines 1–18) includes the interface definition (`Service`) as well as the implementation of the primitive components (`Server` and `Client`) we use in this application. The second part of the descriptor (cf. lines 20–34) focuses on the architecture of the application as a composition of components conforming to the SCA specifications. The composite `Helloworld` therefore declares two primitive components and a property `counter`, whose value is used by the component `Server` (cf. lines 20–34). This part of the definition combines the keywords introduced in Listing 6 to define the structure of the application with the variability points reported in Listing 5 (cf. lines 25–26, 29 & 31) to detail the technologies employed by the application (*e.g.*, Java, SOAP, WSDL, HTTP, JMS).

```
trait Service {
  def print(msg: String)
}

class Server extends Service {
  @Property var count = 0

  def print(msg: String) {
    for (i <- 1 to count) println(msg)
} }

class Client extends Runnable {
  @Property var header: String = _
  @Reference var s: Service = _

  def run {
    s print header+"Hello, world!"
} }

object Helloworld extends Composite("Helloworld") {
  val cnt = property[Int]("counter") is 3

  val srv = new component("Server") {
    property[Int]("count") from cnt
    val s = service("s") exposes Java[Service]
  } uses Bean[Server]
  new component("Client") {
    property[String]("header") is ">> "
    val r = service("r") exposes Java[Runnable]
    reference("s") targets srv.s
  } uses Bean[Client]

  service("run") promotes components("Client").r
}
```

Listing 7: HelloWorld Example.

Discussion. Using FraSCAla, 34 lines of code in Scala are sufficient to describe an architecture and the implementation of an HelloWorld application, which is equivalent to an ArchJava description [1]. The noise introduced by the Scala notation remains limited (37 characters out of the 787 for the whole definition) compared to the flexibility brought by this approach. This flexibility is further illustrated in the next sections. Indeed, the most interesting feature of FraSCAla consists in its capability of extending the aforementioned concepts to define domain-specific or application-specific statements in order to leverage the description of complex software architectures.

3.3 Integrating ADL Patterns

Another example of ADL extension consists in supporting the definition of architectural patterns to leverage the definition of complex architectures. In this section, we therefore provide a couple of examples of architectural patterns that can be programmed out of the SCA language we introduced. These patterns are then included in the base ADL as new statements, which can be immediately used to ease the definition of a complex software architecture.

3.3.1 SCA Intent Pattern

As a first example of architectural pattern, we propose to reflect the concept of SCA *intent* [2] as an SCA component. In SCA, an intent isolates a crosscutting concern, which can then be woven into one or several ports in order to intercept service invocations. This mechanism is expected to be used for implementing non-functional services, such as transaction demarcation or authentication. However, the implementation of such capabilities usually requires the intent to interact with third part services, such as a transaction service or a key store. Therefore, an elegant way for describing an intent consists in adopting a reflective approach and to use SCA components to describe the intent itself [21].

Listing 8 therefore illustrates the definition of a new concept `intent`, which is defined as a composite component promoting a specific service. The implementation of the intent component (whose definition is mentioned as a parameter of the intent) is therefore expected to conform to the interface `IntentHandler` specific to the FraSCAti framework. However, the technology used to implement this intent is left open by the definition of the architectural pattern in order

to accommodate the different programming languages that can be used to realize an intent.

```
case class intent(id: String,impl:ScaImplementation)
   extends Composite(id) {

  val comp = new component(id+"-intent") {
    import org.ow2.frascati.tinfi.api.IntentHandler
    service("intent") exposes Java[IntentHandler]
  } uses impl

  service("intent") promotes comp.services("intent")
}
```

Listing 8: Intent Pattern.

Once defined, the intent pattern becomes available as a new keyword of the ADL and can be directly used to leverage the definition of more complex architectures. Listing 9 provides an example of architecture extension, which uses a legacy architecture definition (described in Listing 7) to introduce a logging intent implemented in Python (cf. line 2). An intent can be declared once and reused in several places. In this example, the logging intent is woven within all the components of the composite component including services and references (cf. line 4), as well as all the services of the surrounding composite component (cf. line 5). Here one could note that the parameters given to the `foreach` method of `Set` are anonymous functions. From this definition, the FraSCAla framework is able to produce the set of artifacts required to deploy this architecture, including not only the architecture descriptor for the extended HelloWorld application, but also the one for the intent definition.

```
object LoggedHelloworld extends Helloworld {
  val log = intent("Logger", Script("logger.py"))

  components("Client") weaves log
  components("Server").services("s") weaves log
  weaves(log)
}
```

Listing 9: Example of Logging Intent.

3.3.2 *Delegation Chain Pattern*

Another example of architectural pattern, which is commonly used in the literature, is the component-oriented version of the delegation chain design pattern [1, 11]. This architectural pattern consists in chaining a list of components that provide and require compatible ports. While this kind of architectural patterns is relatively difficult to describe with a declarative approach, FraSCAla leverages functional programming to ease the definition of such parametric patterns. Listing 10 therefore reports on the definition of the delegation chain in FraSCAla. This pattern takes a list of component implementations as a parameter. The pattern automatically builds and composes the components by connecting the first component to the second and so on.

```
trait DelegationChainPattern extends Composite {
  def delegate(id:String, itf:ScaInterface,
               impls:List[ScaImplementation]):
               List[component] =
    impls match {
      case Nil => Nil
      case impl :: tail => {
        val chain = delegate(id, itf, tail)

        new component(id+"-proxy"+tail.size) {
          service(label) exposes itf
          val del = reference(id+"-delegate")
          del targets chain.head.services(id)
        } uses impl) :: chain
} } } }
```

Listing 10: Delegation Chain Pattern.

Once defined, the resulting architectural pattern can be used within the target architecture as described in Listing 11. This code excerpt extends the Helloworld architecture we defined in Listing 7 to include a delegation chain that filters and decorates the messages that are exchanged between the client and the server. The value `chain` is used to store the list of components that are created by the function `delegate()`. As the components of the delegation chain are automatically linked, the architect does only have to connect the first and the last component of the delegation chain to the client and the server components, respectively.

```
object ChainedHelloworld extends Helloworld
 with DelegationChainPattern {
  val code = List(Bean[MessageFilter], Bean[Decorator])
  val chain = delegate("s", Java[Service], code)

  wire(components("Client").references("s"),
       chain.head.services("s"))
  wire(chain.last.references("s-delegate"),
       components("Server").services("s"))
}
```

Listing 11: Example of Delegation Chain.

Discussion. By exploiting the capabilities of the Scala programming language, FraSCAla supports the definition of parametric architecture templates that can be used to isolate repetitive patterns. The two examples we provide illustrate two different approaches to define architectural patterns. The definition of the delegation chain as a function is recommended when the concepts do not need to be extended and/or the pattern is a compound result (a list of components in the case of the delegation chain). The definition of the intent as a composite component is rather advised when the architect is interested in specializing a concept to introduce dedicated attributes or functions.

4. CASE STUDIES

In this section, we report on two case studies we initiated to assess the benefits of the FraSCAla framework. For the sake of consistency, these case studies are considering SCA-based architectures. Nonetheless, a similar validation could be considered on a different technology. We therefore selected two frameworks which exhibit architecture patterns: the FraSCAti assembly factory (cf. Section 4.1) and the COSMOS context framework (cf. Section 4.2).

4.1 The FraSCATi Case Study

The FraSCAti middleware platform focuses on the development and execution of distributed SCA-based applications [15, 21]. As depicted in Figure 6, FraSCAti is a reflective platform, which is itself built as an SCA application—*i.e.*, its different subsystems are implemented as SCA components. The FraSCAti assembly factory adopts a plugin-based architecture where so called processor components are used to isolate the interpretation of architectural elements. Several plugins are already available in the FraSCAti platform, such as WSDL, SOAP, BPEL, XSD, Java, Script, Java RMI, etc.

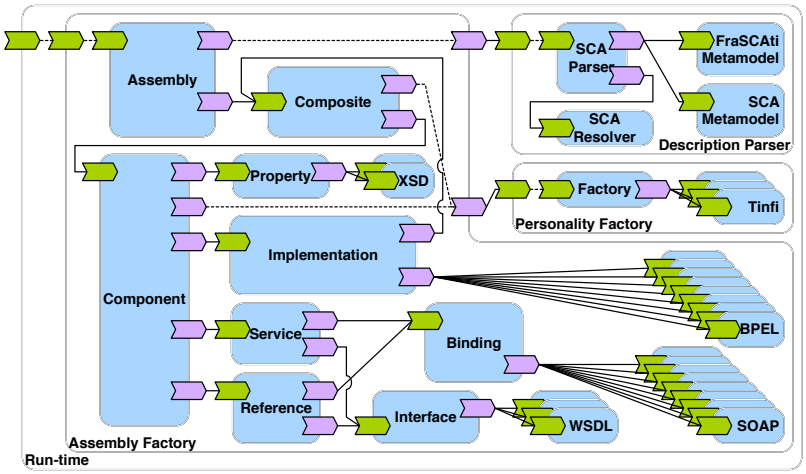

Figure 6: Architecture of the FraSCAti Assembly Factory.

Listing 12 therefore proposes a template architecture for the FraSCAti assembly factory and identifies placeholders, which can be used to tune the configuration of the toolchain. In particular, this template supports the injection of technology-specific processors. As these processors are expected to be developed in Java and to implement the interface `Processor`, the processor pattern is using a type parameter `T` which is used to check the type compatibility of the underlying Java class.

```
class FraSCAti extends Composite("FraSCAti") {                    1
  val dp = new component("Description Parser") {                   2
        /* skipped */                                              3
  }                                                                4
  val af = new component("Assembly Factory") {                     5
    /* skipped */                                                  6
    import org.ow2.frascati.assembly.factory.api._                 7

    case class process[T<:Processor](id:(String,String))           9
      extends component("Processor "+id._2) {                      10
        service(id._1) exposes Java[Processor]                     11
    } uses Bean[T]                                                 12
} }                                                                13
```

Listing 12: Definition of a template architecture of FraSCAti.

Listing 13 therefore describes a specific configuration of the FraSCAti assembly factory that includes the support for Web Services and RESTful technologies. This configuration is described in three parts. The first (cf. lines 1–6) groups all the processors that are concerned with Web Services technologies. The second part (cf. lines 8–13) defines all the processors that cover RESTful services. The third part (cf. line 15) builds a concrete instance of FraSCAti by composing the groups of processors to be used to process the SCA-based architecture descriptions.

```
trait FraSCAtiWS extends FraSCAti {                                1
  af.process[WsdlProcessor]("interface","WSDL")                    2
  af.process[SoapProcessor]("binding","SOAP")                      3
  af.process[BpelProcessor]("implementation","BPEL")               4
  af.process[XsdProcessor]("property-type","XSD")                  5
}                                                                  6

trait FraSCAtiREST extends FraSCAti {                              8
  af.process[WadlProcessor]("interface","WADL")                    9
  af.process[RestProcessor]("binding","REST")                      10
  af.process[HttpProcessor]("binding","HTTP")                      11
  af.process[VeloProcessor]("implementation","Velocity")           12
}                                                                  13

object MyFraSCAti extends FraSCAtiWS with FraSCAtiREST             15
```

Listing 13: Buidling FraSCAti "à la carte".

4.2 The COSMOS Case Study

COSMOS is a component-based framework for managing context information in ubiquitous context-aware applications [6, 19]. COSMOS decomposes context observation policies into fine-grained units called context nodes.

The basic structuring concept of COSMOS is the *context node*. A context node is context information modeled by a software component. COSMOS organizes context nodes into hierarchies to form *context management policies*. The relationships between context nodes are sharing and encapsulation. The sharing of a context node—and, by implication, of a partial or complete hierarchy—corresponds to the sharing of part or all of a context policy. Context nodes at a hierarchy's leaves (the bottom-most elements, with no descendants) encapsulate raw context data obtained from collectors (such as operating system probes, sensors near the device, user preferences in profiles, and remote devices). Context nodes should provide all the inputs necessary for reasoning about the execution context, which is why COSMOS considers user preferences as context data. A context node's role is thus to isolate the inference of high-level context information from lower architectural layers responsible for collecting context data. Figure 7 depicts the mapping from a context policy tree to the associated component-based software architecture. The reader can refer to [6, 19] to get a detailed description of the characteristics of COSMOS.

While COSMOS policies are actually defined using a general purpose ADL called FractalADL [5, 14], we propose to use the FraSCAla framework to build an ADL dedicated to the definition of context policies. For the sake of brevity, Listing 14 reports only on the resulting ADL notation that can be used to describe a context policy. Opposite to the FractalADL approach, which implicitly requires to mapped the concepts of COSMOS into the equivalent component-based constructions, our approach supports the manipulation of the domain-specifc concepts introduced

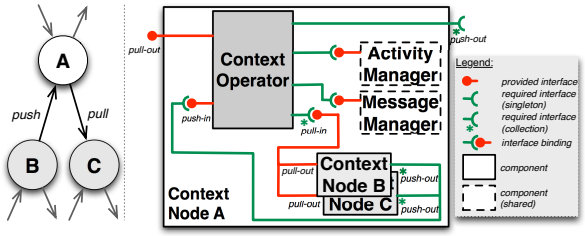

Figure 7: Architecture of COSMOS Context Policy.

by the conceptual model of COSMOS. Therefore, as illustrated in Listing 14, the architect describes the architecture of a context policy using keywords such as `Policy`, `sensor` or `node`, which are more straightforward than generic component-based constructions.

```
object WiFiDetector extends Policy {
  val WiFiMgr = new sensor[WirelessInterfaceRM] {
    parameter("resourceName","eth1")
  } isActiveObserver isBlockingObserver

  val WiFiConnect = new node[ConnectivityDetectorCO] {
    node[AverageCO] consumes WiFiMgr.get("link-quality")

    val AvgWiFiBitRate = new node[AverageIfCO] {
      consumes(WiFiMgr.get("bit-rate", "is-variable"))
  } } isActiveObserver isActiveNotifier

  new node[AdjustedBitRateCO] {
    consumes(WiFiConnect)
    consumes(WiFiConnect.AvgWiFiBitRate)
} }
```

Listing 14: Describing a COSMOS Context Policy.

The approach we propose can therefore be used to quickly prototype a *Domain-Specific Language* (DSL) that is expected to describe component-based architecture. In the case of COSMOS, a DSL dedicated to the description of DSL has been proposed in [19]. However, the design of such a language is a tedious process which requires the specification of a grammar and the implementation of the associated interpreter to convert the context policy descriptions into software architectures. Any modification to be integrated into this DSL would require to reconsider the design of the grammar as well as the implementation of the interpreter, which can be considered as cumbersome in a prototyping phase. Therefore, the approach promoted by FraSCAla can be used to quickly validate a design of a DSL as well as its mapping towards a specific software architecture before initiating the time-consuming development of a more sustainable version.

5. RELATED WORK

General Purpose ADLs. During the last decade, various generic *Architecture Description Languages* (ADLs) have been proposed by the community to ease and improve the quality of software architectures (*e.g.*, Wright, UniCon, Acme, C2, Darwin, Rapide) [16]. As introduced by this paper, some of these ADLs are promoting the definition of architectural styles (also known as architectural patterns) as a support for describing large-scale, potentially heterogeneous and distributed applications. However, the FraSCAla framework we describe goes beyond the simple definition of architectural styles by reflecting the architectural patterns as domain-specific statements that are made available to the architect. Furthermore, we believe that the versatile nature of FraSCAla can accommodate the variety of concepts that are proposed by the literature.

XML-based ADLs. Another widespread approach applied to describe software architectures consists in using XML-based ADLs. Example of such ADLs are xADL [13], xArch [9], FractalADL [14], SCA Assembly Language [2]. By providing an extensible XML-based representation for software architectures, these approaches can be mapped to general purpose ADLs, as demonstrated by xAcme[7], which implements the Acme on top of xArch. However, these approaches promotes the use of declarative approach, which requires the associated semantics to be implemented somewhere else—*i.e.*, in a toolchain—which might become quickly time-consuming for the architect depending on the complexity of the toolchain. Although ADL toolchains, such as FractalADL [14] or FraSCAti [21], provide a modular to facilitate the integration of domain-specific concerns or architecture patterns, the implementation of such semantics remains cumbersome and requires a deep understanding of the underlying technologies. For example, the development of a plugin in the FraSCAti toolchain requires at least the integration of Java, EMF, SCA artifacts. Nonetheless, as we demonstrated in this paper, ADLs prototyped with FraSCAla can easily be converted to XML descriptors conform to one of these approaches in order to quickly prototype new architectural constructions.

Domain-Specific Languages. Another interesting approach covered by the literature refers to the use of *Domain-Specific Languages* (DSL), which provide a domain-oriented notation for building application. In this context, a category of DSL, such as ArchJava [1] or ComponentJ [20], deals with component-oriented programming and propose to merge architectural description with component implementations by specializing the Java programming language with CBSE keywords. However, the definition of these keywords remains hardcoded within the grammar and cannot be further extended. The approach promoted by FraSCAla rather builds on the Scala programming language and injects the architecture description language as an embedded DSL, which brings the capabilities of extending any architectural description with domain-specific constructions.

UML Profiles. Finally, the last category of related work refers to the *Unified Modeling Language* (UML) [3], which includes component-oriented diagrams as well as the concept of *profile* as a mean to specialize general purpose concepts with a specific semantics. The definition of UML profiles can support the definition of architectural patterns and domain-specific constructions, this approach suffers from the same weaknesses as XML-based ADLs, since the operational semantics of the profiles requires to be developed in specific tools (*e.g.*, Rational Rose) in order to be assessed.

6. CONCLUSION

This paper introduced FraSCAla as a framework to rapidly prototype domain-specific architecture description languages. Rapid prototyping is particularly useful to investigate the definition of new architectural constructions without falling

[7] `http://www.cs.cmu.edu/~acme/pub/xAcme`

into time-consuming development processes. FraSCAla therefore provides a versatile and homogeneous approach to design the structural model of an ADL before deriving the associated language. In particular, we illustrate in this paper how FraSCAla can be used to build a generic component model that is then refined towards an ADL for the *Service Component Architecture* (SCA) industrial standard. Based on this language, we demonstrate the capability of FraSCAla to introduce architectural patterns as new statements of the ADL. This capability is assessed on a case study including two SCA-based software, namely the FraSCAti assembly factory and the COSMOS context framework. In both cases, the use of FraSCAla provides an efficient approach to experiment domain-specific architecture description languages without requiring any extensive development since the domain-specific architecture descriptions can be automatically converted to the low-level notations understood by the supporting toolchains (*e.g.*, SCA assembly language, FractalADL, xArch).

In the future, we plan to extend this approach to support the definition and the checking of architectural constraints that need to be enforced by the described software architectures. Additionally, we plan to include the support for expressing the dynamics of software architectures. Given that nowadays software are deployed over long period and need to continuously adapt themselves upon variations in their environment, the description of their semantics should be considered from the initial stages of their design and we believe that FraSCAla can provide a relevant contribution to the state-of-the-art by providing the support for prototyping the definition of reconfiguration and evolution primitives.

Source accessibility. The main source code of the framework is available at: `https://github.com/rouvoy/frascala`.

7. REFERENCES

[1] M. Abi-Antoun, J. Aldrich, D. Garlan, B. R. Schmerl, N. H. Nahas, and T. Tseng. Modeling and implementing software architecture with acme and archjava. In *ICSE*, pages 676–677. ACM, 2005.

[2] Beisiegel, M. et al. Service Component Architecture, 2007. `http://www.osoa.org`.

[3] G. Booch, J. Rumbaugh, and I. Jacobson. *Unified Modeling Language User Guide, The (2nd Edition) (The Addison-Wesley Object Technology Series)*. Addison-Wesley Professional, May 2005.

[4] H. P. Breivold, I. Crnkovic, and M. Larsson. A systematic review of software architecture evolution research. *Information & Software Technology*, 54(1):16–40, 2012.

[5] E. Bruneton, T. Coupaye, M. Leclercq, V. Quéma, and J.-B. Stefani. The fractal component model and its support in java. *Softw., Pract. Exper.*, 36(11-12):1257–1284, 2006.

[6] D. Conan, R. Rouvoy, and L. Seinturier. Scalable processing of context information with cosmos. In *DAIS*, volume 4531 of *LNCS*, pages 210–224. Springer, 2007.

[7] G. Coulson, G. S. Blair, P. Grace, F. Taïani, A. Joolia, K. Lee, J. Ueyama, and T. Sivaharan. A generic component model for building systems software. *ACM Trans. Comput. Syst.*, 26(1), 2008.

[8] I. Crnkovic, J. A. Stafford, and C. A. Szyperski. Software components beyond programming: From routines to services. *IEEE Software*, 28(3):22–26, 2011.

[9] E. M. Dashofy, A. van der Hoek, and R. N. Taylor. A highly-extensible, xml-based architecture description language. In *WICSA*, pages 103–112. IEEE, 2001.

[10] N. Esfahani and S. Malek. On the role of architectural styles in improving the adaptation support of middleware platforms. In *ECSA*, volume 6285 of *LNCS*, pages 433–440. Springer, 2010.

[11] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1 edition, Nov. 1994.

[12] P. Hnetynka and F. Plasil. Using meta-modeling in design and implementation of component-based systems: the sofa case study. *Softw., Pract. Exper.*, 41(11):1185–1201, 2011.

[13] R. Khare, M. Guntersdorfer, P. Oreizy, N. Medvidovic, and R. N. Taylor. xadl: Enabling architecture-centric tool integration with xml. In *HICSS*, 2001.

[14] M. Leclercq, A. E. Özcan, V. Quéma, and J.-B. Stefani. Supporting heterogeneous architecture descriptions in an extensible toolset. In *ICSE*, pages 209–219. IEEE, 2007.

[15] F. Loiret, R. Rouvoy, L. Seinturier, and P. Merle. Software engineering of component-based systems-of-systems: a reference framework. In *CBSE*, pages 61–66. ACM, 2011.

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

[17] M. Odersky. The scala experiment: can we provide better language support for component systems? In *POPL*, pages 166–167. ACM, 2006.

[18] M. Odersky and M. Zenger. Scalable component abstractions. In *OOPSLA*, pages 41–57. ACM, 2005.

[19] R. Rouvoy, D. Conan, and L. Seinturier. Software architecture patterns for a context-processing middleware framework. *IEEE Distributed Systems Online*, 9(6), 2008.

[20] J. C. Seco, R. Silva, and M. Piriquito. Component j: A component-based programming language with dynamic reconfiguration. *Comput. Sci. Inf. Syst.*, 5(2):63–86, 2008.

[21] 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, 2012.

[22] D. Steinberg, F. Budinsky, M. Paternostro, and E. Merks. *EMF: Eclipse Modeling Framework 2.0*. Addison-Wesley Professional, 2nd edition, 2009.

[23] A. Taherkordi, F. Loiret, R. Rouvoy, and F. Eliassen. A generic component-based approach for programming, composing and tuning sensor software. *Comput. J.*, 54(8):1248–1266, 2011.