Painless Support for Static and Runtime Verification of Component-Based Applications

Nuno Gaspar, Ludovic Henrio, Eric Madelaine

Painless support for static and runtime verification of component-based applications

Nuno Gaspar[1,2,3,⋆], Ludovic Henrio[2], and Eric Madelaine[1,2]

[1]INRIA Sophia Antipolis
{Nuno.Gaspar,Eric.Madelaine}@inria.fr

[2]Univ. Nice Sophia Antipolis, CNRS, I3S, UMR 7271, 06900 Sophia Antipolis, France
Ludovic.Henrio@cnrs.fr

[3]ActiveEon S.A.S
http://www.activeeon.com/

Abstract Architecture Description Languages (ADL) provide descriptions of a software system in terms of its structure. Such descriptions give a high-level overview and come from the need to cope with arbitrarily complex dependencies arising from software components.
In this paper we present Painless, a novel ADL with a *declarative* trait supporting parametrized specifications and architectural reconfigurations. Moreover, we exhibit its reliable facet on its integration with ProActive — a middleware for distributed programming. This is achieved by building on top of Mefresa, a Coq framework for the reasoning on software architectures. We inherit its strong guarantees by extracting certified code, and subsequently integrating it in our toolchain.

Keywords: The Coq Proof Assistant, Component-based Engineering, Formal Methods, Architecture Description Language

1 Introduction

Typically, one uses an Architecture Description Language (ADL) as a means to specify the software architecture. This promotes separation of concerns and compels the software architect to accurately define structural requisites. Nevertheless, this task is seldom trivial as arbitrarily complex architectures may need to be defined. It is thus important to provide the means for expressive and intuitive, yet reliable, specifications.

In this paper we present Painless, a novel ADL for describing parametrized software architectures, and its related formal verification support. We discuss its integration with ProActive [1], a middleware for distributed programming, and the reference implementation for the Grid Component Model (GCM) [2].

The GCM ADL lacks support for architectural reconfigurations and parametrization. Further, it is XML-based: while it may be suitable for tools, it is

⋆ This work is partially supported by ANRT/CIFRE No 2012/0109

a rather verbose and static description of the architecture. PAINLESS supports both the definition of parametrized architectures and the specification of reconfigurations in a declarative style. This facilitates deployment tasks and gives a more comprehensive understanding of the application's topology.

For instance, let us consider the motivating example depicted by Figure 1.

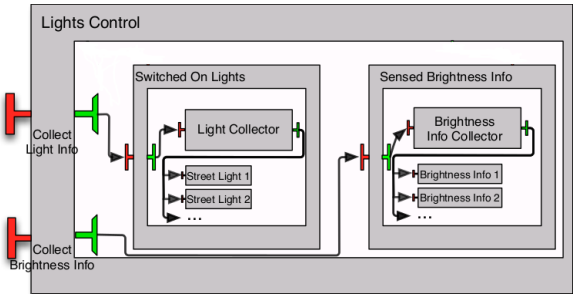

Figure 1. Architecture of the Lights Control use case

This architecture concerns a previous use case [3] on the saving of power consumption by adequately adding/removing Street Light and Brightness Info components. For such scenario an ADL solely describing the deployment topology and unable to handle parametrized specifications becomes cumbersome. In this paper, our main goal is to provide an ADL specifying at the same time the initial deployment and the possible reconfigurations, while providing support for describing parametrized topologies. We also want to rely on formal methods to guarantee a safe deployment and reconfiguration of the considered systems.

In [11], we presented MEFRESA — a Coq [16] framework providing the means for the formal reasoning on software architectures. Here, we extend MEFRESA with the ability to interpret PAINLESS specifications, and provably correct functions computing their compliance with the GCM technical specification [10]. We take advantage of Coq's strong guarantees by extracting certified code, and subsequently integrate it with the ProActive middleware. In our previous work we focused on the mechanization of the GCM, and facilities for developing arbitrarily complex proofs regarding its intricacies. In this paper, we focus on the pragmatical aspects of deployment and reconfiguration by providing an ADL, and all the toolchain that allows us to deploy and reconfigure GCM applications in ProActive while reusing the guarantees provided by our proven back-end.

We see our contribution as two-fold. Firstly, we propose PAINLESS, a novel ADL supporting parametrized specifications and architectural reconfigurations. Its declarative nature promotes concise and modular specifications. Secondly, we describe the integration of its related tool support with ProActive. This provides a case study on the use of certified code, fostering the application of formal methods in a software engineering context.

The remainder of this paper is organised as follows. Section 2 briefly discusses GCM and Mefresa. Section 3 overviews our approach for extending the ProActive middleware to cope with Painless specifications. Section 4 introduces the semantics of Painless. Section 5 shows the specification of the use case depicted by Figure 1 in Painless. Related work is discussed in Section 6. For last, Section 7 concludes this paper.

2 Background

Mefresa provides a mechanized specification of the GCM, a simple *operation* language for manipulating architectural specifications, and the means to prove arbitrary complex properties about instantiated or parametrized architectures. It is developed with the Coq proof assistant [16].[1]

The GCM is constituted by three core elements: interfaces, components, and bindings.

An interface is defined by an *id* denoting its name, a *signature* corresponding to its *classpath*, and a *path* identifying its location in the component's hierarchy (i.e. the component it belongs to). It is of internal or external *visibility*, has a client or server *role*, is of functional or non-functional *functionality*, has an optional or mandatory *contingency*, and its *cardinality* is singleton, multicast or gathercast.

A component has an *id*, a *path*, a *class*, subcomponents, interfaces, and bindings. This implicitly models GCM's hierarchical nature. Further, components holding subcomponents are called *composite*.

Bindings act as the means to connect components together through their interfaces. They are composed by a *path* indicating the component holding the binding, and *id*s identifying the involved components and interfaces. Moreover, they can be of normal, import or export kind. A normal binding connects two components at the same hierarchical level, that is, they have the same enclosing component. The remaining kind of bindings are connecting together a component with a subcomponent. Whether of import and export kind depends on the client interface being from the subcomponent or from the enclosing one, respectively.

The GCM technical specification [10] dictates the constraints that a GCM application must comply with. They can be summed up into properties regarding the *form* of the architecture and its readiness to start execution. These requirements are encoded by the well-formed and well-typed predicates.

well-formed and well-typed architectures A component is well-formed if its subcomponents are well-formed and uniquely identifiable through their identifiers. Further, its interfaces, and bindings must also be well-formed.

Interfaces are well-formed if they are uniquely identifiable by their identifiers and visibility value: two interfaces may have the same identifier provided that they

[1] Mefresa is available online at `http://mefresa.gaspar.link`

have a different visibility. bindings are well-formed if they are established between existing components/interfaces, from client to server interfaces, and unique.

A component may be well-formed but still unable to start execution. Further insurances are needed for the overall good functioning of the system in terms of its application dependencies. These are dictated by typing rules (see [10, p. 22]).

An interface possesses cardinality and contingency attributes. These determine its supported communication model and the guarantee of its functionality availability, respectively. For instance, for proper system execution we must ensure that client and singleton interfaces are bound at most once. For client interfaces only those of multicast cardinality are allowed to be bound more than once.

Analogously, similar constraints apply to the interfaces' contingency attribute. An interface of mandatory contingency is guaranteed to be available at runtime. This is rather obvious for server interfaces as they implement one or more service methods, i.e., they do have a functionality of their own. Client interfaces however, are used by service methods that require other service methods to perform their task. It therefore follows that a client and mandatory interface must be bound to another mandatory interface of server role. As expected, interfaces of optional contingency are not guaranteed to be available.

MEFRESA captures these requirements by defining a well-typed predicate. Basically, it requires that both the contingency and cardinality concerns are met throughout the component hierarchy. Architectures not meeting these requirements are said to be *ill-typed*.

An operation language for manipulating GCM architectures Another important element of MEFRESA is an *operation* language that allows the manipulation of GCM architectures. It possesses seven constructors: Mk_component, Rm_component, Mk_interface, Mk_binding, Rm_binding, Seq, and Done. The meaning of each constructor should be intuitive from its name. The only doubt may arise from the Seq constructor: it stands for operation composition.

Its operational semantics is mechanized by the step predicate, and exhibit the following structure: $op \;/\; \sigma \rightarrow op' \;/\; \sigma'$. States are denoted by σ, and in our particular case these have the shape of a component, i.e., an empty state is an empty component, etc. Thus, σ represents the component hierarchy being built.

With Coq, one can use these semantic rules to interactively reduce an operation to its *normal form* done, at which point some final state σ is attained. Naturally, the ability to perform such reduction depends on the demonstration that all required premises for each individual reduction step are met. This lets us wonder about a more general property that one can expect about σ on an overall operation reduction. Let $\longrightarrow$* be the reflexive transitive closure of the step predicate. Then, the theorem depicted by Listing 1.1 should be intuitive.

```
1 Theorem validity: forall (s s':state) (op:operation),
2     well_formed s          ->
3     op / s --->* Done / s' ->
4     well_formed s'.
```

Listing 1.1. validity statement

Informally, it expresses that if s is a well-formed state, and if we are able to reduce op to Done, then we know that the resulting state s' is well-formed. Proving this theorem is achieved by induction on the operation language constructors.

3 Overview of our approach

Figure 2 gives an overview of our approach. In short, we obtain an extension to ProActive that is able to cope with PAINLESS architectures.

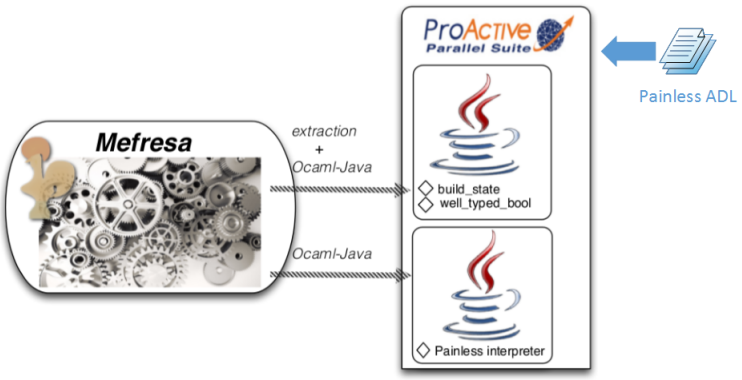

Figure 2. Integration of Painless with ProActive

We extend MEFRESA with functions — build_state and well_typed_bool — responsible for ensuring the compliance of a deployment/reconfiguration specification with the GCM requirements. We prove these functions correct w.r.t the GCM mechanized specification, and use Coq's extraction mechanism to obtain certified OCaml code. Further, we also define a PAINLESS interpreter that translates PAINLESS expressions to MEFRESA's operation language. This is directly programmed in OCaml. Finally, to ease the integration with ProActive, we use OCaml-Java [5] to produce Java byte code.

3.1 Painless hello world

PAINLESS provides the software architect with the ability to write parametrized architectures and its possible structural reconfigurations in a declarative style. An excerpt of its grammar is defined by Table 1.

Its elementary — or *normal forms* — expressions include natural numbers, booleans, lists, and strings. Naturally, one can also use variables. Making and removing elements from the component architecture is achieved by the polymorphic mk and rm, respectively. As expected, skip is idempotent. Components, interfaces and bindings are also first-class citizens — where bexp is an expression

exp ::= n | true | false | [] | *str* | x | mk **exp** | rm **exp** | skip
| Component **exp**$_1$ ... **exp**$_6$ | Interface **exp**$_1$ ... **exp**$_7$ | Binding **bexp**
| **exp** :: **exp** | **exp** = **exp** | **exp** + **exp** | **exp** − **exp**
| if **exp** then **exp** else **exp** | **exp exp** | **exp** ; **exp**
| match **exp** with **pat**$_1$ → **exp**$_1$ ... **pat**$_k$ → **exp**$_{k\geq 1}$ end

decl ::= let P = **exp** | let rec P = **exp**

rcfg ::= Reconfiguration *str* arg$_0$ ... arg$_k$**: decl**$_0$ ... **decl**$_k$ reconfigure **exp**

arch ::= Architecture *str* **: decl**$_0$ ... **decl** $_{k\geq 0}$ deploy **exp rcfg**$_0$ ... **rcfg**$_{h\geq 0}$

Table 1. PAINLESS syntax (excerpt)

for the three types of bindings. Facilities for manipulating lists, comparison, and binary operators such as + and - are also built-in features. The standard if-then-else, function application, sequence **;** and match constructors conclude the range of allowed expressions. **decl** acts as a declaration layer composed by the usual (potentially recursive) let definitions, indexed by a parameter *P*.

An architecture **arch** is composed by a string str representing its name, $k \geq 0$ declarations, and an expression describing the application deployment topology. Further, it may contain $h \geq 0$ similarly defined reconfigurations.

Listing 1.2 depicts a simple PAINLESS specification.

```
1 Architecture "Street Light component":
2 let itf_class = "org.lightscontrol.GetLightInfo"
3 let impl_class = "org.lightscontrol.StreetLight"
4
5 let itf = Interface "GetLightInfo" itf_class ["Street Light"] External
      Server Functional Mandatory Singleton
6 let streetLight = Component "Street Light" [] impl_class [] [itf] []
7 deploy mk streetLight
```

Listing 1.2. A first Painless specification

Its meaning should be intuitive. We give a representative name to the specification (line 1), and define two definitions holding an interface and component class (lines 2-3). Then, we define an interface named *"GetLightInto"*, using the previously defined class, with a path indicating the component it belongs, and followed by its attributes concerning its visibility, role, etc (line 5). Next, we define the component named *"Street Light"*, with an empty path — i.e., at the root of the component hierarchy —, with impl_class as its implementation class, without subcomponents, with itf as its only interface, and without bindings (line 6). Finally, we deploy the application (line 7).

3.2 Computing states from operations

PAINLESS specifications are translated to MEFRESA's operation language. The details of this process are discussed in Section 4.

As discussed above, one can check the feasibility of reducing an operation by interactively applying its reduction rules and attempting to prove the required

premises. This ability is of great value when attempting to prove arbitrary complex properties about parametrized architectures. Yet, if we intend to build a state representing the result of an operation reduction, then we would be better with a function performing such task. This is the purpose of the function depicted by Listing 1.3.

```
1  Function build_state (op:operation) (s:state) : option state :=
2   match op with
3   | Mk_component i p ic cl lc li lb =>
4     if beq_bool (valid_component_path_bool p s && no_id_clash_bool i p s
5        && dec_component (Component i p ic cl lc li lb)) false then
6         None
7     else
8        add_component s p i ic cl lc li lb
9    ...
10
11  | Seq          op1 op2            =>
12       match build_state op1 s with
13         | None    => None
14         | Some s' => build_state op2 s'
15       end
16  | Done                            => Some s
17  end.
```

Listing 1.3. Excerpt of the build_state function definition

The above excerpt shows how we can use a function to compute the result of an arbitrary operation reduction. Basically, it pattern matches on the parameter op (line 2), and proceeds depending on the matched constructor. For instance, if it is a Mk_component, it performs the adequate checks w.r.t. to the creation of a component, and invokes the add_component function (lines 3-8). As expected, valid_component_path_bool is a boolean function checking if path p points to an existing component in the state init. no_id_clash checks that the identifier i is not already used by another component at the same hierarchical level. For last, dec_component computes whether the component to be added is well-formed.

Apart from the Seq and Done constructors, the remaining operation constructors are handled analogously. Seq is composed by two operations (line 11), the leftmost operation is fully evaluated, and the resulting state is used for evaluating the rightmost operation (lines 12-15). Done means that the end of the operation was reached, and it simply returns the current state (line 16).

Another important note regards the use of the option type as return type of this function. This is due to the fact that it only returns a state if it was able to fully evaluate the given operation, otherwise, if the operation is *invalid*, it simply returns None. As seen above, the validity theorem (see Listing 1.1) enunciates that reducing an operation to Done from a well-formed state yields a well-formed state. Naturally, the analogous behaviour is expected from the build_state function. Further, we also expect it to always be able to compute a resulting state from an operation op, whenever it is possible to fully evaluate op. Formally, listing 1.4 depicts the relevant theorem.

```
1  Theorem build_state_correctness:
2    forall op s s',
3      well_formed s ->
4      (op / s --->* Done / s' <-> build_state op s = Some s').
```

Listing 1.4. build_state correctness

Proving build_state_correctness requires a case analysis on the operation constructors, and relating the boolean checks made in build_state with the premises of the step predicate.

Considering the context of a component-based application life-cycle, one deploys its application by performing an operation *op* on an empty state — which is provably well-formed. Then, if *op* can indeed be reduced, we reach a well-formed state s (see Listing 1.1). Performing an architectural reconfiguration boils down to applying an operation *op*' to s, leading to yet another well-formed state s' — provided that *op*' can indeed be reduced —, and this can be repeated indefinitely. Indeed, there is no need to explicitly compute the well-formedness of the attained states, as it is provably guaranteed. There is however such a need regarding their well-typedness. To this end, we define the $well\_typed\_bool : component \rightarrow boolean$ function. Basically, it acts as a decision procedure w.r.t. the well-typedness of a component. It is proved as the computational counterpart of the well_typed predicate, that is, it is both *sound* and *complete* w.r.t. the well_typed predicate.

If an issue occurs — invalid operation or ill-typedness of the returned state — an exception is thrown and the deployment aborts. Otherwise, the operation is mapped to the adequate methods composing the ProActive API, and the actual deployment is performed by the middleware. Further, the object holding the state's structure is kept for subsequent reconfiguration tasks.

4 Painless semantics

Table 2 gives an excerpt of the rules for translating expressions to Mefresa's operation language. We use $\Gamma \vdash e \Downarrow v$ for denoting the evaluation of e under the environment Γ being reduced to v, and $\vdash_t$ stands for type inference.

Rule nf_{sem} dictates that a *normal form* yields immediately a semantic value. The rule $skip_{sem}$ simply depicts that skip is translated to Mefresa's done operation. Rules mk^c_{sem} and mk^i_{sem} illustrate the polymorphic constructor **mk** at work. It can be used to build components, interfaces and bindings — making bindings is omitted for the sake of space. These proceed by fully reducing the expression e into a component/interface/binding that can be used into Mefresa's operations. Rule c_{sem} shows the reduction of a Component: all its elements (identifier, subcomponents, ...) need to be reduced and of adequate type. Analogous rules apply for Interfaces and Bindings. $match_{sem}$ illustrates how pattern matching is performed. First, the expression *exp* to be matched is reduced to some value *val*. Then, we reduce the expression exp_k with the corresponding pattern $pat_{k\in\{1,n\}}$ matching with *val*. As expected, this occurs in an environment Γ enlarged with a mapping between pat_k and *val*, and patterns are checked by order. var_{sem} shows that a variable is reduced by looking it up in the environment Γ. Finally, the rule seq_{sem} simple attests that a sequence of Painless expressions is translated to Mefresa's operations.

$$\frac{normal\_form(v)}{\Gamma \vdash v \Downarrow v}\ nf_{sem} \qquad \frac{}{\Gamma \vdash skip \Downarrow \textbf{Done}}\ skip_{sem}$$

$$\frac{\Gamma \vdash e \Downarrow c \quad c = \textbf{\textit{Component}}\ id\ p\ cl\ lc\ li\ lb}{\Gamma \vdash mk\ e \Downarrow \textbf{Mk\_component}\ c}\ mk^{c}_{sem} \qquad \frac{\Gamma \vdash e \Downarrow i \quad i = \textbf{\textit{Interface}}\ id\ p\ sig\ v\ f\ co\ ca}{\Gamma \vdash mk\ e \Downarrow \textbf{Mk\_interface}\ i}\ mk^{i}_{sem}$$

$$\frac{\begin{array}{c}\Gamma \vdash exp_1 \Downarrow id \quad \Gamma \vdash_t id : string \quad \Gamma \vdash exp_2 \Downarrow p \quad \Gamma \vdash_t p : list\ string \\ \Gamma \vdash exp_3 \Downarrow cl \quad \Gamma \vdash_t cl : string \quad \Gamma \vdash exp_4 \Downarrow lc \quad \Gamma \vdash_t lc : list\ component \\ \Gamma \vdash exp_5 \Downarrow li \quad \Gamma \vdash_t li : list\ interface \quad \Gamma \vdash exp_6 \Downarrow lb \quad \Gamma \vdash_t lb : list\ binding\end{array}}{\Gamma \vdash Component\ exp_1\ ...\ exp_6 \Downarrow \textbf{Component}\ id\ p\ cl\ lc\ li\ lb}\ c_{sem}$$

$$\frac{\Gamma \vdash exp \Downarrow val \quad matches(pat_k, val) \wedge \forall h, h < k \rightarrow \neg matches(pat_h, val) \quad \Gamma, (pat_k, val) \vdash exp_k \Downarrow v_k}{\Gamma \vdash match\ exp\ with\ pat_1 \rightarrow exp_1\ ...\ pat_n \rightarrow exp_n\ end \Downarrow v_k}\ match_{sem}$$

$$\frac{\Gamma[x] = \alpha}{\Gamma \vdash x \Downarrow \alpha}\ var_{sem} \qquad \frac{\Gamma \vdash exp_1 \Downarrow \alpha \quad \Gamma \vdash_t \alpha : operation \quad \Gamma \vdash exp_2 \Downarrow \beta \quad \Gamma \vdash_t \beta : operation}{\Gamma \vdash exp_1\ ;\ exp_2 \Downarrow \alpha\ ;\ \beta}\ seq_{sem}$$

Table 2. PAINLESS semantic rules (excerpt)

The complete reduction of an expression should yield a (sequence of) MEFRESA's operations, otherwise it is rejected. For instance, the rule $arch_{sem}$ depicts how an architecture without reconfiguration strategies is evaluated.

$$\frac{\begin{array}{c}\forall i, 0 \leq i \leq k.\ decl_i = (P_i,\ exp_i) \quad \Gamma \vdash exp_i \Downarrow \beta_i \\ \Gamma, (P_0, \beta_0), ..., (P_k, \beta_k) \vdash exp \Downarrow \alpha \quad \Gamma \vdash_t \alpha : operation\end{array}}{\Gamma \vdash Architecture\ str\textbf{: decl}_0\ ...\ \textbf{decl}_{k \geq 0}\ deploy\ \textbf{exp} \Downarrow \alpha}\ arch_{sem}$$

Basically, the deployment expression *exp* is reduced to α, under an environment including all the declarations $decl_i$. Naturally, α must be of type *operation*.

Dealing with reconfigurations is performed analogously. The expression to be evaluated is reduced on a context including the deployment declarations, the ones defined locally, and its instantiated parameters.

4.1 Painless standard library

As discussed above, the GCM component model is hierarchical, that is, a component may possess subcomponents. A component communicates with the "outside" world through its *external* interface, whereas it relies on its *internal* interfaces to communicate with its subcomponents. Typically, composite component interfaces are symmetric, that is, for each external interface of server role there is a internal interface of client role, and vice-versa. Listing 1.5 and Listing 1.6 depict a convenient function to ease the specification of such scenarios — with the obvious definition of visibility_symmetry omitted for the sake of space.

```
let role_symmetry r =
 match r with
   Client -> Server
 | Server -> Client
 end
```

Listing 1.5. Role symmetry

```
let symmetric i = match i with
   Interface id si p v r f co ca ->
     let vs = visibility_symmetry v in
     let rs = role_symmetry r in
     Interface id si p vs rs f co ca end
```

Listing 1.6. Interface symmetry

Another common scenario regards the need to change the location of a component. For this, we define the function depicted by Listing 1.7.

```
1  let change_component_path p comp =
2    match comp with
3      Component id cp cl lc li lb ->
4
5        let rec change_subcomponents_path p lc =
6          match lc with
7            []      -> []
8          | c :: r -> change_component_path p c ::
                       change_subcomponents_path p r
9          end
10       in
11
12       let lcm = change_subcomponents_path (suffix p id) lc in
13       let lim = change_interfaces_path (suffix p id) li in
14       let lbm = change_bindings_path (suffix p id) lb in
15       Component id p cl lcm lim lbm
16   end
```

Listing 1.7. Changing the path of a component

A component may contain subcomponents, interfaces and bindings. As such, it is also necessary to adjust their paths. We define a inner function (lines 5-9) to deal with nested recursion. The function suffix returns a path with the second parameter suffixed to the first one. Moreover, we use other library functions — change_interfaces_path and change_bindings_path — to adjust the interfaces and bindings paths (lines 13-14).

Another useful function concerns the making of components in a specific path. Listing 1.8 defines such a function.

```
let mk_in p c =   mk (change_component_path p c)
```

Listing 1.8. Changing the path of a component

All the discussed functions are part of Painless standard library along with other facilities for dealing with common specification tasks. Further, the user can easily build its own libraries as specifications can be imported.

5 Specifying the Lights Control use case in Painless

In this section we show how the specification of the Lights Control application discussed in Section 1 (see Figure 1) is achieved in Painless. We follow a modular approach by separately specifying the Switched On Lights, Sensed Brightness Info, and Lights Control components.

Listing 1.9 depicts the specification of the Switched On Lights component.

```
1  Architecture "Composite component: Switched On Lights":
2
3  let id = "Switched On Lights"
4  let p  = [id]
5
6  let streetLight =
7    Component "Street Light" p "org.lightscontrol.StreetLight"
8    []
9    [Interface "GetLightInfo" "org.lightscontrol.GetLightInfo"
10    [id; "Street Light"] External Server Functional Mandatory Singleton]
11   []
12
13 let collectLightInfo p =
14   Interface "CollectLightInfo" "org.lightscontrol.CollectLightInfo"
15         p External Server Functional Mandatory Singleton
16
17 let getLightInfo =
18   Interface "GetLightInfo" "org.lightscontrol.GetLightInfo"
19      [id; "Light Collector"] External Client Functional Mandatory
           Multicast
20
21 let lightCollector =
22   Component "Light Collector" p "org.lightscontrol.LightCollector"
23   [] [collectLightInfo [id; "Light Collector"] ; getLightInfo] []
24
25 let switchedOnLights nrOfStreetLights =
26   Component id [] "null"
27   (lightCollector :: list_of streetLight nrOfStreetLights)
28   [collectLightInfo p ; symmetric (collectLightInfo p)]
29   (Export p "CollectLightInfo" "Light Collector" "CollectLightInfo" ::
30    normal_bindings p "Light Collector" "GetLightInfo" "Street Light" "
         GetLightInfo" nrOfStreetLights )
```

Listing 1.9. Specification for the Switched On Lights component (from Figure 1)

We start by giving a descriptive name to this ADL (line 1). Then, we define the Street Light component (lines 6-11). It possesses a name, a path indicating where it is in the component hierarchy, a classpath, an empty list of subcomponents, one server interface and no bindings. This definition should be seen as a template, as its instances are the ones dynamically added/removed. Next, we define the Light Collector component (lines 21-23) and its two interfaces (lines 13-19). The first interface is parametrized by its path as we shall use it later when specifying the Lights Control component (see Listing 1.11). Last, we specify the Switched On Lights component parametrized by its number of Street Lights (lines 25-30). As expected, its subcomponents include the Light Collector component and a list of *nrOfStreetLights* Street Light components (line 27). The interfaces are symmetric and their specification is conveniently handled by the interface_symmetry function. Further, the function normal_bindings is responsible for binding Light Collector's multicast interface to the several Street Light instances.

It should be noted that this specification can be used on its own by adding a deployment expression. Listing 1.10 depicts an example of a deployment with one hundred Street Light components.

```
33 deploy mk (switchedOnLights 100)
```

Listing 1.10. Example of a deployment specification for Switched On Lights component

The ADL of the Sensed Brightness Info component follows the same rationale and is omitted for the sake of space. Listing 1.11 depicts the deployment spe-

cification of the overall Lights Control application. As an example, the Street Light and Sensed Brightness Info components are instantiated to ten each.

```
1  Require "org.lightscontrol.adl.SwitchedOnLights.painless"
2  Require "org.lightscontrol.adl.SensedBrightnessInfo.painless"
3  Architecture "Lights Control Architecture":
4
5  let p = ["Lights Control"]
6
7  let lightsControl =
8    Component "Lights Control" [] "null" []
9      [collectBrightnessInfo p; symmetric (collectBrightnessInfo p);
10      collectLightInfo p      ; symmetric (collectLightInfo p)      ] []
11
12 let n = 10 //number of sensor components to deploy
13 let m = 10 //number of light components to deploy
14
15 deploy
16  mk (add_subcomponents lightsControl [sensedBrightnessInfo n;
       switchedOnLights m]);
17  mk export p "CollectLightInfo" "Switched On Lights" "CollectLightInfo";
18  mk export p "CollectBrightnessInfo" "Sensed Brightness Info" "
       CollectBrightnessInfo"
```

Listing 1.11. Specification for the Lights Control application

We start by importing the ADLs from the Switched On Lights and Sensed Brightness Info components (lines 1-2). This adds all their definitions to the current scope, namely the interfaces collectLightInfo and collectBrightnessInfo. Next, we define the Lights Control without including its subcomponents and bindings (lines 7-10). These are added directly in the deployment expression. The function add_subcomponents belongs to PAINLESS standard library. It places the subcomponents into LightsControl while adequately adjusting their path field (line 16). Finally, the two export bindings are established to the two added subcomponents (lines 17-18).

The last remaining ingredient concerns the structural reconfigurations. Listing 1.12 depicts two reconfiguration strategies regarding the addition and removal of the n^{th} Street Light component.

```
22 Reconfiguration "add light" n:
23   let p = ["Lights Control" ; "Switched On Lights"]
24   reconfigure
25     mk_in p (nth streetLight n);
26     mk normal p "Light Collector" "GetLightInfo" ("Street Light"+n) "
         GetLightInfo"
27
28 Reconfiguration "remove light" n:
29   let p = ["Lights Control" ; "Switched On Lights"]
30   reconfigure
31     rm normal p "Light Collector" "GetLightInfo" ("Street Light"+n) "
         GetLightInfo" ;
32     rm ["Lights Control" ; "Switched On Lights"] ("Street Light"+n)
```

Listing 1.12. Reconfigurations specification for the Lights Control application

Their understanding should pose no doubt. The first adds a Street Light component by making it with the adequate path (line 25) and subsequently binding it to Light Collector's multicast interface (line 26). As expected, the expression nth streetLight n returns a streetLight component with an identifier suffixed by n. The second reconfiguration is handled in a similar manner. We first need to

unbind the component to remove (line 31) — where normal is the constructor for normal bindings —, and then we proceed by removing it (line 32).

From a programming perspective, the reconfigurations are available through a simple method call indicating its name and parameters. Further, the evaluation of the deployment specification and subsequent applied reconfigurations is carried out by the machinery originating from MEFRESA. Moreover, it should be noted that checking that a reconfiguration leads to a well-formed and well-typed component architecture is achieved without stopping any component. Indeed, before reconfiguring the application, ProActive needs to stop the involved composite component. The inherent benefit is that only valid reconfigurations w.r.t the mechanized GCM specification are mapped to the ProActive API. For instance, attempting to add a Street Light component with the same identifier as another one already deployed is rejected, i.e., an exception is thrown.

Our ProActive extension is freely available online. The release contains the examples discussed here and several others. The reader is pointed to the following website for more details `http://painless.gaspar.link`.

6 Related Work

Let us mention the work around the ArchWare ADL [14]. They claim that *"software that cannot change is condemned to atrophy"* and introduce the concept of an *active software architecture*. Based on the higher-order π-calculus, it provides constructs for specifying control flow, communication and dynamic topology. Unlike PAINLESS, its syntax exhibits an imperative style and type inference is not supported, thus not promoting concise specifications. Nevertheless, it is sufficiently rich to provide executable specifications of active software architectures. Moreover, user-defined constraints are supported through the ArchWare Architecture Analysis Language. Yet, their focus is more aimed at the specification and analysis of the ADL, rather than actual application execution and deployment. In our work, the user solely defines the architecture of its application, structural constraints are implicit: they are within the mechanized GCM specification. Further, our tool support is tightly coupled with ProActive.

Also from the realm of process algebras, Archery [15] is a modelling language for software architectural patterns. It is composed by a core language and two extensions: Archery-Core, Archery-Script and Archery-Structural-Constraint. These permit the specification of structural and behavioural dimensions of architectures, the definition of scripts for reconfiguration, and the formulation of structural constraints, respectively. Moreover, a bigraphical semantics is defined for Archery specifications. This grants the reduction of the constraint satisfaction verification to a type-checking problem. However, this process is not guarantee to be automatic, and type-checking decidability remains as future work.

Gerel [9] is a generic reconfiguration language including powerful query constructs based on first-order logic. Further, its reconfiguration procedures may contain preconditions *à la* Hoare Logic [12]. These are evaluated by brute force. It is unclear how they cope with the inherent undecidability of such task.

Di Cosmo et. al. defined the Aeolus component model [6]. Their focus is on the automation of cloud-based applications deployment scenarios. Their proposal is loosely inspired by the Fractal component model [4] whose most peculiar characteristics are its hierarchical composition nature and reconfiguration capabilities. However, while both approaches permit architectural reconfigurations at runtime, its specification is not supported by their ADL, it solely contemplates deployment related aspects. Moreover, support for parametrized specifications is also not covered, forcing the software architect to explicitly define the application's structure.

Regarding Fractal, it is also worth noticing that it tries to overcome the lack of support for reconfiguration specification through Fscript [7]. Fscript embeds FPath — a DSL for navigation and querying of Fractal architectures — and acts as a scripting language for reconfiguration strategies. These are not evaluated for their validity. Nevertheless, system consistency is ensured by the use of *transactions*: a violating reconfiguration is *rolled back*.

Like the Fractal ADL, xMAML [13] is XML-based, yet it permits the specification of reconfigurations. An important difference is that their focus is on processor architectures and aim at producing synthesizable models.

In [8], Di Ruscio et. al. defend the concept of building your own ADL through the byADL framework. Further, they claim that *"it is not possible to define a general, optimal ADL once and forever"*, and propose the means to incrementally extend and customize existing ADLs by composing their metamodels. This approach offers an interesting perspective regarding the interoperability of Painless with other ADLs.

7 Final Remarks

In this paper we presented Painless and its related novel approach for the specification of software architectures. Its declarative trait allows for intuitive and concise specifications, liberating the software architect from highly verbose specifications such as the ones obtained via machine languages like XML. Moreover, its support for parametrized architectures eases deployment — it becomes a matter of instantiation —, and thus boosts productivity. Further, in ProActive, mapping components to physical resources is achieved through *application/deployment descriptors*. While this information is not an aspect of the architecture *per se*, extending Painless with such feature could be envisaged.

Another key ingredient is the treatment of structural reconfigurations as first-class citizens. Indeed, by supporting the specification of the topological changes that may occur at runtime, it yields a better understanding of the application. Moreover, it is worth noticing that the specified reconfigurations become easily accessible from a programming perspective: through a simple method call with the name of the desired reconfiguration. Furthermore, reconfiguration specifications are evaluated at runtime. The clear benefit is that one can be highly confident that the reconfiguration will not leave the application in a *ill formed* or *ill typed* state as the evaluation process is carried out by provably correct code

extracted from Mefresa. Additionally, a further inherent advantage is that it all happens without stopping the application. Indeed, actually performing the reconfiguration requires it to be stopped at the composite level. By making a prior evaluation, the risk of reconfiguration failure is avoided.

References

1. ActiveEon S.A.S. ProActive - A Library for Parallel and Distributed Programming.
2. Françoise Baude, Denis Caromel, Cédric Dalmasso, Marco Danelutto, Vladimir Getov, Ludovic Henrio, and Christian Pérez. GCM: a grid extension to fractal for autonomous distributed components. *Annales des Télécommunications*, 2009.
3. Françoise Baude, Ludovic Henrio, and Paul Naoumenko. Structural reconfiguration : an autonomic strategy for GCM components. In *Proc. of The Fifth International Conference on Autonomic and Autonomous Systems: ICAS 2009*, 2009.
4. Eric Bruneton, Thierry Coupaye, and Jean-Bernard Stefani. The Fractal component model, 2004.
5. Xavier Clerc. OCaml-Java: OCaml on the JVM. In Hans-Wolfgang Loidl and Ricardo Peña, editors, *Trends in Functional Programming*, volume 7829 of *LNCS*, pages 167–181. Springer, 2012.
6. Roberto Di Cosmo, Stefano Zacchiroli, and Gianluigi Zavattaro. Towards a formal component model for the cloud. In George Eleftherakis, Mike Hinchey, and Mike Holcombe, editors, *SEFM*, volume 7504 of *LNCS*, pages 156–171. Springer, 2012.
7. Pierre-Charles David, Thomas Ledoux, Thierry Coupaye, and Marc Léger. FPath and FScript: Language support for navigation and reliable reconfiguration of Fractal architectures. *Annals of Telecommunications*, Volume 64(Numbers 1-2 / February 2009):45–63.
8. Davide Di Ruscio, Ivano Malavolta, Henry Muccini, Patrizio Pelliccione, and Alfonso Pierantonio. ByADL: An MDE framework for building extensible architecture description languages. In MuhammadAli Babar and Ian Gorton, editors, *Software Architecture*, volume 6285 of *LNCS*, pages 527–531. Springer, 2010.
9. M. Endler and J. Wei. Programming generic dynamic reconfigurations for distributed applications. *Configurable Distributed Systems, 1992., International Workshop on*, pages 68–79, 1992.
10. ETSI. ETSI TS 102 829 V1.1.1 - GRID; Grid Component Model (GCM); GCM Fractal Architecture Description Language (ADL). Technical Spec., ETSI, 2009.
11. Nuno Gaspar, Ludovic Henrio, and Eric Madelaine. Bringing Coq into the world of GCM distributed applications. *International Journal of Parallel Programming*, pages 1–20, 2013.
12. C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, Volume 12(Number 10):pages 576–580, 1969.
13. Julien Lallet, Sébastien Pillement, and Olivier Sentieys. xMAML: A modeling language for dynamically reconfigurable architectures. In Antonio Núñez and Pedro P. Carballo, editors, *DSD*, pages 680–687. IEEE Computer Society, 2009.
14. Ronald Morrison, Graham N. C. Kirby, Dharini Balasubramaniam, Kath Mickan, Flávio Oquendo, Sorana Cîmpan, Brian Warboys, Bob Snowdon, and R. M. Greenwood. Constructing Active Architectures in the ArchWare ADL. *CoRR*, 2010.
15. Alejandro Sanchez, Luís Soares. Barbosa, and Daniel Riesco. Bigraphical modelling of architectural patterns. In Farhad Arbab and Peter Csaba Ölveczky, editors, *FACS'2011*, volume 7253 of *LNCS*, pages 313–330. Springer, 2011.
16. The Coq Development Team. The Coq Proof Assistant Reference Manual, 2012.