Fractal à la Coq

Nuno Gaspar, Eric Madelaine

Fractal *à la* Coq

Nuno Gaspar*
Oasis Project Team
INRIA Sophia Antipolis - Méditerranée
Nuno.Gaspar@inria.fr

Eric Madelaine
Oasis Project Team
INRIA Sophia Antipolis - Méditerranée
Eric.Madelaine@inria.fr

Abstract

Component-based Engineering aims at providing a modular means to specify a wide range of applications. The idea is to promote a clean separation of concerns, and thus reusability, in order to ease the burden of software development and maintenance. The specification of such component models however, tends to be informal, leaving their inherent ambiguities open to interpretation.

In this paper we present our ongoing work towards a formal specification of the Fractal Component Model mechanized in the Coq Proof Assistant. An operational semantics for building component-based architectures is presented, along with its compliance with the Fractal specification.

1 Introduction

Building software is becoming a more and more complex task. Our era of computers and electronics demands highly sophisticated applications, where not only performance but also reliability are expected. Coping with such needs requires special care when designing software. The approach promoted by Component-based Engineering tackles these requests by leveraging the view of software as *building blocks* that when put together form the intended functionality. This is the rationale followed by the Fractal Component Model [**?**].

Essentially, the essence of the Fractal Component Model lies around the notions of *component*, *interface* and *binding*. The components represent entities, generally pieces of software code, that communicate explicitly and necessarily by means of interfaces. This is in fact the main difference between Component-based programming and standard Object-oriented programming. The inherent advantage is that the dependencies between software blocks become explicit, permitting an easier software maintenance. At last, bindings convey procedures calls or messages passing between interfaces.

Components are deemed *composite* if they possess subcomponents. These can be shared, thus possibly yielding graph architectures and not only tree structures. Moreover, monitoring and *on the fly* reconfiguration of applications are other concerns tackled by Fractal.

1.1 Motivation and Related Work

Many approaches regarding the formalization of component models can be found in the literature. While it is still an utopia to see formal methods as part of the standard software engineering discipline, the maturity attained by formal tools already permit considerable achievements.

In [**?**], Henrio et al. present a framework for reasoning on component composition mechanized in Isabelle/HOL. Their focus is on a GCM-like component model[1] and show promising results for dealing with reconfiguration properties in a mechanical way.

*The author is partially funded under contract CIFRE 2012/0109 ActiveEon/INRIA

[1]The Grid Component Model (GCM)[**?**] is an extension of Fractal for distributed and parallel architectures.

Moreover, a formal specification of the Fractal Component Model has been proposed in the Alloy specification language [**?**]. This work proves the consistency of a (set-theoretic) model of Fractal applications. This is perhaps the work most closely related with ours.

Their specification however, is constrained by the first-order relational logic nature of the Alloy Analyzer. In fact, they point to the use of the Coq Proof Assistant in order to overcome this limitation.

In this paper we present our work in progress towards a mechanization of the Fractal Specification in the Coq Proof Assistant. Our approach differs from the remaining in that we aim at providing a semantics that allows the building of *correct-by-construction* Fractal architectures.

The remainder of this paper is organized as follows. Section **??** shows the encoding of Fractal's basic definitions. Next, section **??** details a semantics for building Fractal Architectures. The kind of properties that can be proved in our development are discussed in section **??**. Finally, section **??** concludes by pointing some directions for future work.

2 Mechanizing Fractal Architectures

As mentioned above, the Fractal Component Model has three core elements: components, interfaces and bindings. As such, we shall consider a simple language built around these core elements. In the following, we omit the obvious definitions for the sake of space.

2.1 Basic Definitions

Let us first demonstrate how interfaces and components are mechanized in the Coq Proof Assistant.

```
Inductive interface : Type :=
  | Interface     : ident         -> (*its id*)
                    type          -> (*its type*)
                    path          -> (*path to the component it belongs*)
                    accessibility -> (*Internal/External*)
                    communication -> (*client/server*)
                    functionality -> (*functional/Control*)
                    language      -> (*the language it implements*)
                    interface.
```

An interface is characterized by an *identifier*, a *type*, a *path* identifying the component where the interface belongs to, whether it is accessible internally or externally, whether is it supposed to communicate as client or server, whether it serves functional or control purposes, and finally the language it implements. The intended meaning of each of these fields should be clear. The only complex aspect may arise from the *path* field. Fractal is a hierarchical component model, and since by introspection an interface is able to identify the component it belongs to, a *path* identifying this component is necessary. Its definition is a list of identifiers, where these identifiers indicate the components that need to be *traversed* in the hierarchy to reach the component holding the interface.

```
Inductive component : Type :=
  | Component          : ident          -> (*its unique id*)
                         type           -> (*its type*)
                         path           -> (*its path on the hierarchy*)
                         controlLevel   -> (*Lowest/Introspection/Configuration*)
                         list component -> (*its sub components*)
                         list interface -> (*its interfaces*)
                         list binding   -> (*its bindings*)
                         component.
```

A component has an *identifier*, a *type*, a *path* indicating its level in the hierarchy, a control level determining what it can do, subcomponents, interfaces and bindings. Bindings are connecting components together by means of their interfaces.

```
Inductive binding : Type :=
  | Normal : path -> ident -> ident -> ident -> ident -> binding
  | Export : path -> ident -> ident -> ident -> binding
  | Import : path -> ident -> ident -> ident -> binding.
```

A binding is always established between a client and a server interfaces. It can be a *normal*, *export* or *import* binding. These are defined by indication of the component's *path* which they belong to, and by the identifiers of the involved components and interfaces. For export and import bindings the identifier of one intervening component can always be inferred and thus is omitted.

2.2 Operational Semantics of Fractal Architectures

Having defined our core elements, it is now time to introduce a notion of *state*. A state is the structure that holds all the relevant information. For instance, for a tiny programming language it is generally a function mapping variables to their values.

```
Definition state := component.

Definition empty_state : state :=
        Component (Ident "Root") Top nil Configuration nil nil nil.
```

In our case, a state has the same shape of a component. An empty state is therefore a component named *Root* without any subcomponents, interfaces and bindings.

The syntactic categories of our language for building Fractal Architectures are defined as follows.

$$
\begin{array}{rcl}
\text{a} & ::= & \quad \textbf{mk\_component} \text{ component} \\
& & \mid \textbf{mk\_interface} \text{ interface} \\
& & \mid \textbf{bind} \text{ binding} \\
& & \mid \text{a; a} \\
& & \mid \textbf{done}
\end{array}
$$

The meaning of the first three constructs should raise no doubt: making components, making interfaces and establishing bindings. We use ***a*** for representing *actions*[2]. Allowing for multiple actions to execute as a sequence is defined by means of the standard operator **;**. Last, **done** stands for the completed action.

The design of software architectures can be seen from a transition system point of view. One makes some action **a**, in some state σ, and ends up with a reduced action **a'** in some state σ**'**. This can be represented by the following manner.

$$\langle a, \sigma \rangle \longrightarrow \langle a', \sigma' \rangle.$$

Building Fractal architectures will therefore require to define these transition rules for each constructor of our language. In other words, to define a semantics.

In the following we use the **.** notation for projections.

[2]In the realm of programming languages semantics, **instruction** or **statement** are the terms usually employed. In our case we consider that **action** is a more adequate nomenclature.

Making Components

$$\frac{\begin{array}{l} c = Component\ id\ t\ p\ cl\ subComps\ interfaces\ bindings \\ valid\_component\_path\ p\ \sigma \\ well\_formed\_component\ c \\ \forall\ c',\ c' \in (get\_scope\ p\ \sigma) \rightarrow (c'.id \neq id) \end{array}}{\langle make\_component\ c,\ \sigma\rangle \longrightarrow \langle done,\ add\_component\ \sigma\ c\rangle}\ (SMakeComponent)$$

The above rule dictates that for creating a component one must supply a valid path, it must be itself well formed, and its identifier must be distinct from the ones in the same scope. The **valid_component_path** predicate expresses the *well-formedness* of a path in the hierarchy of components for the current state. Component *well-formedness* is inductively defined in Coq as follows.

```
Inductive well_formed_component (c: component) : Prop :=
  | WellFormedComponent_Base:
        well_formed_interfaces (c->interfaces)                                  ->
        well_formed_bindings (c->bindings) (c->subComponents) (c->interfaces) ->
        (c->subComponents) = nil                                                ->
        well_formed_component c
  | WellFormedComponent_Step:
        (forall c', In c' (c->subComponents) -> well_formed_component c')       ->
        unique_ids (c->subComponents)                                           ->
        well_formed_interfaces (c->interfaces)                                  ->
        well_formed_bindings (c->bindings) (c->subComponents) (c->interfaces) ->
        well_formed_component c.
```

As for interfaces, they are deemed well formed provided that their pair of values **ident** $\times$ **accessibility** is unique. Moreover, each binding must be *valid*, that is, be of normal, import or export kind. These predicates are defined analogously to the precedent.
At last, it is requested that inserting a component will not break the unicity of identifiers in the hierarchical level being inserted to. The function $get\_scope : path \rightarrow state \rightarrow list\ component$ returns the list of components where the insertion is supposed to occur.

Making Interfaces

$$\frac{\begin{array}{l} i = Interface\ id\ t\ p\ a\ c\ f\ l \\ valid\_interface\_path\ p\ \sigma \\ c = get\_component\_with\_path\ p\ s \\ \forall\ i',\ i' \in (c.interfaces) \rightarrow i'.id = id \rightarrow i'.accessibility \neq a \end{array}}{\langle make\_interface\ i,\ \sigma\rangle \longrightarrow \langle done,\ add\_interface\ \sigma\ i\rangle}\ (SMakeInterface)$$

As in the creation of components, creating an interface also requires a valid path. The function $get\_component\_with\_path : path \rightarrow state \rightarrow component$ returns the component where the interface is supposed to be added. Inserting the interface must not break the property of unique pair of values **ident** $\times$ **accessibility**.

Making Bindings

$$\frac{\begin{array}{l} valid\_component\_path\ (b.path)\ \sigma \\ valid\_component\_binding\ b\ components \end{array}}{\langle make\_binding\ b,\ \sigma\rangle \longrightarrow \langle done,\ add\_binding\ \sigma\ b\rangle}\ (SMakeBinding)$$

As in the above rules, a valid path is expected. Further, establishing a binding also requires the **valid_component_binding** predicate to hold. As stated in the Fractal specification, a binding is either a normal, export or import binding.

```
Inductive valid_component_binding (b:binding) (s:state) : Prop :=
  | NormalBinding: normal_binding b s -> valid_binding b s
  | ExportBinding: export_binding b s -> valid_binding b s
  | ImportBinding: import_binding b s -> valid_binding b s.
```

For instance, a binding is deemed normal if it is established between two external interfaces, and the components owning these interfaces have the same enclosing component, i.e., they are in the same composite component. Moreover, a binding must be established between *client* to *server* interfaces.

Sequence of Actions Finally, our last action serves the purpose of composing actions. Its definition is straightforward: having a composition of an action a_1 with an action a_2, we first need to reduce a_1 so we can reach a_2.

$$\frac{\langle a_1, \sigma\rangle \longrightarrow \langle a_1', \sigma'\rangle}{\langle a_1; a_2, \sigma\rangle \longrightarrow \langle a_1'; a_2, \sigma'\rangle}\ (SSeq_1) \qquad \frac{\langle a_1, \sigma\rangle \longrightarrow \langle done, \sigma'\rangle}{\langle a_1; a_2, \sigma\rangle \longrightarrow \langle a_2, \sigma'\rangle}\ (SSeq_2)$$

So far, all the above rules defined a *one step* transition. However, there are cases where we may want to reason about *multiple steps* transitions. For instance, the action

```
Definition SimpleArchitecture : action :=
 Mk_component nil (Ident "A") SomeType Configuration nil nil;
 Mk_component nil (Ident "B") SomeType Configuration nil nil;
 Mk_interface ((Ident "A")::nil) (Ident "X") SomeType External Client Control C;
 Mk_interface ((Ident "B")::nil) (Ident "Y") SomeType External Server Control C;
 Bind ((Ident "A")::nil) (Ident "X") External
      ((Ident "B")::nil) (Ident "Y") External.
```

requires five steps to complete. Here, what we would like to do is to be able to reason on the overall execution of this action. As such, we need to formalize this notion of multiple steps. This is achieved by the reflexive transitive closure of our transition relation. Considering $step : action * state \rightarrow action * state \rightarrow Prop$, our inductive definition modelling our transition rules, we can define the notion of multiple steps in the following manner.

```
Notation "a '/' s '--->*' a' '/' s'" := (refl_step_closure step (a,s) (a',s')).

Lemma well_formed_architecture:
  exists s,
    SimpleArchitecture / empty_state --->* Done / s.
Proof.
  ...
Qed.
```

Having this notion defined we can now reason on the overall execution of a sequence of actions. For instance, the above trivial lemma states that the execution of our **SimpleArchitecture** will indeed terminate.

2.3 Meeting the Specification

The rigorous definition of the possible actions that one can take allow us to formally reason on all possible outcomes. One important theorem is to prove that starting an architecture in a *well formed* state, when the execution completes it will end up in a state that is also *well formed*.

```
Definition well_formed (s:state) : Prop :=
  match s with
    | Component id t p cl lc li lb =>
      id = (Ident "Root") /\ t = Top /\ p = nil /\ cl = Configuration /\
      li = nil /\ well_formed_component (Component id t p cl lc li lb)
  end.

Theorem validity:
   forall a s s', a / s --->* Done / s'   ->
                  well_formed s          ->
                  well_formed s'.
```

It should be noted that the above theorem is an universal quantification over all possible Fractal architectures. Thus, every architecture built with our semantics rules and obeying the *well-formedness* hypothesis will be well formed.

Preliminary results allow us to conclude on the feasibility of this approach. At this stage however, the proof is not completed and a more precise definition of our **well_formed** predicate is needed for a full compliance with the Fractal specification.

3 Final Remarks

In this paper we presented the first steps towards a mechanized specification of the Fractal Component Model. At this stage, there is still some remaining work to be included in our Coq development. Inclusion of a *progression* theorem and typing rules on bindings are natural next steps. Furthermore, a seamless integration with the Vercors platform [**?**] could be envisaged. In the long term one of our goals is to use this formalization in conjunction with *model checking* techniques in order to address behavioural properties of complex component architectures.

Moreover, our *action* language opens interesting perspectives w.r.t. reconfiguration capabilities of component-based systems. Another research direction to follow would be to see how could we best leverage this kind of approach with the concerns of autonomic reconfigurations.

Acknowledgements

The authors are grateful to Ludovic Henrio for numerous remarks in earlier versions of this paper and engaging discussions about possible future research directions.