Safe Dynamic Reconfigurations of Fractal Architectures with FScript[*]

Pierre-Charles David[†] Thomas Ledoux

OBASCO Group, EMN / INRIA, Lina
`pcdavid@gmail.com`, `ledoux@emn.fr`

Abstract

In this paper we present FScript, a Domain-Specific Language used to program structural reconfigurations of Fractal architectures. Compared to the use of the standard Fractal APIs in a general purpose language, FScript offers better syntactic support for navigation, more dynamicity, and guarantees on the reconfigurations (termination, atomicity, consistency, and isolation). FScript introduces a new notation, called FPath, which is designed to express queries on Fractal architectures, navigating inside them and selecting elements according to predicates. It also provides access to all the primitive reconfiguration actions available in Fractal, and enables the user to define custom reconfigurations using a simple imperative language.

1 Introduction

One of the main features of the Fractal component model [1] is that it is fully dynamic and reflexive: it is possible both to *discover* the structure of a Fractal application (introspection) and to *modify* it (intercession) at run-time. This makes it possible to build, for example, administration tools like Fractal Explorer with which it is easy to navigate inside a running application and modify it interactively. It is also possible to program dynamic reconfigurations, even unanticipated ones [4], to be executed in a running application. This is important in order to evolve applications without stopping and redeploying them (for example to update a component or subsystem) and to build self-adaptive and autonomic systems [3] which must take reconfiguration decisions – and apply them – dynamically and automatically (i.e. without human intervention).

Such dynamic discovery and reconfigurations can be programmed in the same language than the application itself, for example Java, using the standard Fractal APIs. However, such an approach has several drawbacks:

- The Fractal APIs are designed to be minimalist and orthogonal, which is good for tool writers but can be cumbersome and lead to verbose code when used to program specific reconfigurations. This can be offset by building higher-level libraries of helper functions in the general-purpose language, but such extensions are non-standard and must be developed, debugged, and maintained by each user.

[*]This research is supported by the RNTL project Selfware (`http://www.rntl.org/projet/resume2005/self-ware.htm`).

[†]This work has partially been done while the author was a post-doctoral fellow at France Telecom R&D.

- Fractal introduces new concepts like interfaces and bindings, which are not integrated at the host language level, especially in the syntax[1]. This can lead to confusion, for example in Java where two related but distinct concepts of interfaces coexist (Java interfaces and component interfaces): Fractal interfaces are represented by Java objects which implement both a language-level interface and the `Interface` interface...
- Although Java is a relatively dynamic language, which supports dynamic code loading, it is not lightweight and dynamic enough to support dynamic definition and execution of unanticipated reconfigurations in a simple way. To apply an unanticipated Fractal reconfiguration to a running application one would need to: *(i)* write the actual reconfiguration using the Fractal API, with the issues stated above, *(ii)* compile this code, which requires a JDK and access to the running application class files for type checking, *(iii)* deploy the class files on the host system and load it in the JVM running the target application, and *(iv)* finally execute the code.
- Finally, Java being a general purpose language, it is not possible to offer guarantees when executing a Fractal reconfiguration programmed in Java. For example, nothing prevents the reconfiguration code to access and corrupt (intentionally or not) private data structures, to call dangerous method (`System.exit()`), or simply to loop forever.

In order to overcome these limitations while still retaining Fractal's advantages, we have designed and implemented a new language, called FScript, to navigate inside Fractal architectures and dynamically reconfigure them [2]. FScript can be though of as a complement to the standard Fractal ADL[2]: while the ADL (Architecture Description Language) uses a *declarative* approach to specify the *initial* configuration of an application, FScript is an *imperative* language and is used to *incrementally reconfigure* a running application.

To do this, FScript provides a special notation called FPath to navigate intuitively inside an architecture and select parts of it (Section 2). These elements can then be acted upon to reconfigure the architecture using primitive Fractal operations or user-defined reconfigurations scripts (Section 3). Beyond its direct syntactic support for Fractal concepts, the main feature of FScript is that it provides guarantees on the consistency of the reconfigurations by considering these as transactions.

2 Navigation With the FPath Notation

FPath is a special notation used inside the FScript language to *navigate* inside Fractal architectures and *select* elements in it according to some predicate. Its syntax and execution model are inspired by the XPath language [6] which solves the same problem on XML documents (although FPath does not use XML at all).

FPath sees a given Fractal architecture as an oriented graph with labelled arcs. Different kinds of nodes represent all the architectural elements we chose to reify: the *components* themselves (not reified as such in Fractal, but only through the `component` interface); component *interfaces* (both external and internal); configuration *attributes*, corresponding to getter/setter methods on `attribute-controllers`; and finally *methods* on the interfaces. These nodes are connected by labelled arcs, which denote the kind of relation between them. For example, an arc labelled `interface` goes from a given component node to each interface node representing

[1]This is partly due to the lack of flexibility in most languages' syntaxes, as the Fractal specification does not forbid implementations to extends the host language's syntax if possible.

[2]`http://fractal.objectweb.org/fractaladl`

the component's external interfaces. In the same way, if composite C_1 contains C_2 as a sub-component, the corresponding nodes N_1 and N_2 will be connected by two arcs: $N_1 \xrightarrow{\texttt{child}} N_2$ and $N_2 \xrightarrow{\texttt{parent}} N_1$. The following types of arcs, called *axes*, are defined in FPath:

- `component`: from any kind of node to the component owning this node;
- `attribute`: from a component node to all its configuration attributes;
- `interface`: from a component node to all its interfaces, and from a method node to the interface of which it is part;
- `method`: from an interface to all its methods;
- `binding`: from an client interface node to the server interface it is bound to, if any;
- `child` (resp. `parent`): from a component to its direct children (resp. parents); `descendant` (resp. `ancestor`) are the corresponding transitive closures, selecting all the indirect children (resp. parents);
- `sibling`: from a component to all the other components which have at least one direct super-component in common with it.

Given this representation, FPath expressions denote *relative paths* starting from an initial set of nodes in the graph. Such a path is a series of steps, each made of up to three elements: `axis::test[predicate]` (the predicate is optional). On each step, an initial set of nodes is converted to a new set by following all the arcs with a label corresponding to the axis, then filtering the result using the *test* (on the node names) and optional *predicates* (boolean expressions applied to each candidate). For a multi-step path, this algorithm is repeated with the result of the current step as the starting node-set of the next.

For example, the FPath expression `./sibling::*/interface::*[client(.)][not(bound(.))]` is made of two steps. The first one uses the `sibling` axis, an "empty" test `*` (which is always true) and has no predicate. The second step uses the `interface` axis, no test either, and two predicates which are combined. Inside the predicates, the dot "`.`" represents the current node on which the predicate is evaluated. Evaluating the complete expression starting from an initial component node will first select all its sibling components, however they are named, then select all the external interfaces of these siblings, and filter this set of interfaces to return only client interfaces (`client()`) which are not already bound.

The expressions used as predicates can be any FPath expression, which includes not only paths but also standard arithmetic operations, comparisons, function calls, literal strings and numbers and finally variable references (`$varName`). When a path expression is used as a predicate, it is considered *true* if and only if it returns a non-empty set of nodes. For example, to find all the components in a application which provide configuration attributes, one could use the following expression on the application's root component: `descendant-or-self::*[./attribute::*]`.

3 FScript Reconfigurations

The FPath notation described above can be used to navigate inside a Fractal architecture, but not to modify the architecture. The complete FScript language, of which FPath is just a part, enables the definition of *reconfiguration actions* to apply to a running application. Here is a simple example of such a definition which illustrates all of FScript constructs.

```
action auto-bind(comp) {
  // Selects the interfaces to connect
  clients = $comp/interface::*[client(.)][not(bound(.))];
  for itf : $clients {
    // Search for candidates compatible interfaces
    candidates = $comp/sibling::*/interface::*[compatible($itf, .)];
    if ($candidates) {
      // Connect one of these candidates
      bind($itf, one-of($candidates));
    }
  }
  return $comp/interface::*[client(.)][not(bound(.))];
}
```

This defines a new reconfiguration action named `auto-bind`, which takes one parameter, named `comp`, and automatically connects its required interfaces by discovering the compatible server interfaces on sibling components. The body of the action consists in a sequence of simple statements (assignments, procedure calls and return) ended with semicolons and control structures (iteration and conditionals).

FScript distinguishes two kinds of procedures: functions and actions. Functions are guaranteed to be side-effect free, and can only introspect an architecture, not modify it. They can be used safely inside FPath requests, for example in the predicates. Functions are defined like actions, except that they use the `function` keyword instead of `action`, and can only invoke other functions, not actions (be they primitive or user defined). FScript provides a standard library of primitive functions and actions which gives the user access to all the information available from the Fractal API, and all the standard reconfigurations, and new ones can be added easily.

Variables in FScript are not typed (values are). They are created either explicitly on their first assignment ("`varName = expression;`") or implicitly when entering inside the body of a procedure, where parameters behave like local variables. Variable reference is done by preceding the name of the variable with a dollar sign ("`$varName`"). Variables are lexically scoped, but only procedure bodies introduce a new scope, not conditionals and iterations.

The control structures available in FScript are voluntarily limited so that we can guarantee the termination of all reconfigurations:

- *Conditionals* use the classical "`if/then/else`" structure.
- Bounded *iteration* have the syntax "`for i : expression { body }`", where `i` is the name of the iteration variable to use, `expression` is an FPath expression which must return a set of nodes, and `body` is a sequence of statements.
- A procedure can *return a value* immediately and stop its execution by using "`return expression;`".

FScript's design and implementation guarantee the consistency of reconfigurations. Because these reconfigurations are applied to running applications, we must guarantee that they will not break the target system. To this end, we have chosen a set of consistency criterion, in particular *transactional integrity* (atomicity, consistency of the final state, isolation) and *termination* of the reconfigurations. The validation of these criteria is guaranteed in part by the language's structure itself, whose expressive power has been limited, and in part by the implementation. More precisely:

- The definition of (directly or indirectly) recursive actions is forbidden, and the only control structure available for iteration, a `for` loop, iterates on the result of an FPath expression, which always returns a finite set of nodes. These constraints guarantee actions' *termination*, although they do not provide a time bound.
- During the execution of a reconfiguration, the language interpreter keeps a complete journal of all the primitive actions performed, together with enough information to revert them. As soon as an error occurs, the interpreters uses this journal to roll-back the current reconfiguration and return to the initial state. Given that all the primitive Fractal reconfigurations are themselves atomic and reversible, this guarantees the *atomicity* of FScript reconfigurations.
- At the end of a reconfiguration, the interpreter checks that the current configuration is consistent, i.e. that all the required client interfaces are correctly bound to a corresponding server interfaces and that all the components which have been temporarily stopped during the reconfiguration can safely be restarted. If this is not the case, the interpreters cancels the reconfiguration and rolls back to the initial state, thus ensuring the consistency of the application.
- Finally, the *isolation* of reconfigurations is currently guaranteed by globally serialising them. This works but is highly sub-optimal and may be enhanced in future works.

4 Conclusion

We have presented FScript, a Domain-Specific Language [5] to program structural reconfigurations of Fractal architectures. Compared to the use of the standard Fractal APIs in a general purpose language, FScript offers better syntactic support for navigation, more dynamicity, and guarantees on the consistency of the reconfigurations.

FScript introduces a new notation, called FPath, which is designed to express queries on Fractal architectures, navigating inside them and selecting elements according to predicates. FPath is embedded inside the FScript language, but can also be used as a standalone query language. FScript provides access to all the primitive reconfiguration actions available in Fractal, and enables the user to define custom reconfigurations using a simple procedural language. The language is designed and implemented so that it can offer guarantees (termination and transactional integrity) on the reconfigurations..

The language is currently implemented in Java as a simple interpreter, which can be used either through a simple API or using a text-based interactive console (shell). It is available as part of the Fractal project on ObjectWeb[3].

References

[1] E. Bruneton, Th. Coupaye, M. Leclercq, V. Quema, and J.-B. Stefani. An open component model and its support in java. In *Intl. Symp. on Component-Based Software Engineering (CBSE 2004)*, volume 3054 of *LNCS*, pages 7–22, Edinburgh, may 2004. Springer-Verlag.

[2] Pierre-Charles David. *Développement de composants Fractal adaptatifs : un langage dédié à l'aspect d'adaptation.* PhD thesis, Université de Nantes / École des Mines de Nantes, July 2005.

[3] `http://fractal.objectweb.org/fscript`

[3] Jeffrey Kephart. A vision of autonomic computing. In Richard P. Gabriel, editor, *Onward! proceedings from an OOPSLA 2002 track*, pages 13–36, Seattle, WA, USA, November 2002. ACM.

[4] Barry Redmond and Vinny Cahill. Supporting unanticipated dynamic adaptation of application behaviour. In *Proceedings of ECOOP 2002*, volume 2374 of *LNCS*, pages 205–230, Malaga, Spain, May 2002. Springer-Verlag.

[5] Arie van Deursen, Paul Klint, and Joost Visser. Domain-specific languages: An annotated bibliography. *ACM SIGPLAN Notices*, 35(6):26–36, June 2000.

[6] World Wide Web Consortium. XML path language (XPath) version 1.0. W3C Recommendation, November 1999. `http://www.w3.org/TR/xpath`.